Enhancing SAR Satellite Constellation Efficiency Through NATS-Enabled Inter-Satellite Communication

University of Turku
Department of Computing
Master of Science (Tech) Thesis
March 2026
Emad Bin Abid

UNIVERSITY OF TURKU
Department of Computing

Emad Bin Abid: Enhancing SAR Satellite Constellation Efficiency Through NATS-Enabled Inter-Satellite Communication

Master of Science (Tech) Thesis, 142 s.
March 2026

This thesis investigates the feasibility and benefits of inter-satellite communication within a small satellite constellation by developing a proof-of-concept implementation using the Flight Software (FSW) Application Protocol over NATS messaging on top of TCP/IP. The work abstracts away the physical inter-satellite communication layer - such as radio frequency or optical links - and instead assumes the existence of an intermittent IP-based network, where TCP connections between satellites can be established sporadically. This assumption reflects a realistic and reproducible laboratory environment using flatsats.

The primary objective is to evaluate the applicability of software-defined messaging infrastructure in a space-relevant context, with a focus on demonstrating key operational use cases. These include: coordinating imaging tasks across satellites; relaying large data objects from satellites lacking ground connectivity; enabling data uploads from ground to orbit via intermediate satellites; and maintaining synchronized state across nodes through NATS JetStream stream replication.

The system is implemented in Golang, with NATS providing the messaging layer to facilitate resilient and asynchronous communication under constrained and unreliable network conditions. Through controlled testing, the study assesses performance indicators such as message delivery reliability, latency, data throughput, and the robustness of stream synchronization mechanisms. The results illustrate the potential of lightweight, application-layer protocols to support distributed coordination in satellite constellations, contributing to the broader field of software-defined space systems and laying groundwork for future inter-satellite networking architectures.

Keywords: Inter-satellite communication, Flight Software (FSW) Application Protocol, NATS messaging, Satellite constellation, Software-defined space systems

Acknowledgements

This thesis would not have been possible without the guidance, support, and expertise of several individuals who contributed to its completion.

I would like to express my sincere gratitude to my university supervisor, Tuomas Mäkilä, for his invaluable guidance throughout this research. His insightful feedback, thoughtful questions, and encouragement to pursue rigorous experimental validation helped shape this work into a comprehensive investigation. His patience in reviewing multiple iterations and his ability to identify areas requiring deeper exploration were instrumental in elevating the quality of this thesis.

I am deeply grateful to my company supervisor, Markku Kontio, Staff Engineer in the Flight Software team at ICEYE, for providing the industrial perspective and practical context that grounded this research in real-world satellite operations. His extensive experience in satellite flight software and distributed computing informed the architectural decisions and operational scenarios explored in this work. The technical discussions, evaluation sessions, and his willingness to share insights from ICEYE's operational satellite missions enriched my understanding of the challenges facing modern satellite constellations. His support in providing the research environment and resources necessary for this investigation made the experimental work possible.

The opportunity to conduct this research at the intersection of academic rigor and industrial relevance has been invaluable. The combination of academic supervi-

sion from the University of Turku and industry mentorship from ICEYE created an ideal environment for investigating practical solutions to real operational challenges.

Finally, I acknowledge the open-source communities behind NATS and the broader distributed systems ecosystem whose work provided the technological foundation for this research.

Espoo, March 6, 2026

[Emad Bin Abid]

Contents

1 Introduction

The increasing availability and affordability of small satellites, often referred to as CubeSats or microsatellites, has revolutionised the way in which space missions are designed, launched, and operated. Instead of relying on a single large spacecraft, modern space endeavours increasingly deploy constellations of small satellites to achieve persistent coverage, rapid revisit times, and distributed sensing. This shift introduces new opportunities for collaboration between satellites, particularly when multiple modalities of observation are combined. Satellites can generate rich, multi-perspective data products that far exceed the capability of any single satellite platform. However, for such constellations to realise their full potential, it is crucial that satellites not only transmit their data to the ground but also communicate directly with one another in orbit.

Traditional satellite operations rely heavily on the ground segment as the central node for coordination and data relay. Satellites downlink their observations to ground stations, where mission control makes decisions and uplinks further tasking. While this approach is effective, it introduces significant latency, dependency on ground station visibility, and bandwidth bottlenecks. In time-sensitive scenarios, such as monitoring fast-developing events or opportunistic targets, these limitations may render the collected data less useful or even obsolete. Direct satellite-to-satellite communication over inter-satellite links (ISLs) provides a promising avenue to overcome these challenges by enabling distributed, autonomous decision-making within

the constellation.

1.1 Motivation

The motivation for studying inter-satellite communication stems from the desire to enhance autonomy, responsiveness, and resilience of satellite constellations. By allowing satellites to share information and coordinate tasks directly in orbit, several critical benefits can be realised:

- **Reduced latency:** Satellites can exchange situational data and react to events in real time, without waiting for intermittent ground passes.
- **Improved resource utilization:** Tasking can be dynamically redistributed across the constellation, ensuring that the most suitable satellites are allocated to each observation opportunity.
- **Resilience in contested environments:** In cases where ground connectivity is limited, denied, or degraded, satellites can still coordinate among themselves to continue mission objectives.
- **Scalability:** As constellations grow in size, centralised control becomes increasingly inefficient. Peer-to-peer coordination offers a scalable alternative.
- **Data relay and dissemination:** Satellites without direct ground connectivity can leverage their peers to downlink information, ensuring wider coverage and continuity of service.

The emerging concept of software-defined space systems provides a compelling context for implementing these capabilities. By abstracting the communication and coordination mechanisms into software-defined protocols and middleware, satellite

operators gain flexibility and adaptability without being tightly constrained by hardware limitations. In this context, **NATS**, a lightweight and high-performance messaging system, emerges as a candidate for facilitating inter-satellite communication.

1.2 Background and Context

NATS is an open-source messaging system designed to support distributed applications with a focus on simplicity, performance, and scalability. Its publish-subscribe communication model enables decoupled interaction between components, while NATS JetStream provides persistence, message replay, and stream synchronisation features. These properties align well with the requirements of distributed satellite systems operating in constrained and intermittent networking environments.

In this thesis, the physical ISL technology itself—whether radio frequency or optical—is abstracted away. Instead, it is assumed that an intermittent IP network exists between satellites, allowing TCP/IP connections to be established sporadically. This abstraction simplifies experimentation and enables future reproducible testing. By focusing on the application-layer messaging protocol rather than the physical link, the study can isolate and evaluate the feasibility, benefits, and limitations of using NATS in space-relevant conditions.

1.3 Research Questions and Goals

The primary goal of this thesis is to investigate the feasibility of using NATS as a messaging layer for inter-satellite communication, with emphasis on distributed coordination within a heterogeneous constellation. To achieve this, the study develops and evaluates a proof-of-concept system simulating inter-satellite connectivity constraints.

The research questions driving this work are as follows:

1. **Feasibility:** Can NATS, originally designed for terrestrial distributed systems, be effectively applied in a space-relevant context characterised by intermittent connectivity, variable latency, and constrained bandwidth?

2. **Performance:** What are the reliability, latency, and throughput characteristics of NATS messaging under simulated inter-satellite link conditions?

3. **Use case demonstration:** How can inter-satellite messaging be used to enable collaborative operational scenarios, such as tasking satellites for coordinated imaging, relaying data via peers, and synchronising distributed state?

4. **Resilience:** To what extent does NATS support robustness in the presence of intermittent connections and node failures?

5. **Scalability:** How well does the system scale when extended from a few satellites to larger constellations?

In order to address these research questions, the following concrete goals are defined:

- Design and implement a proof-of-concept system where three satellites communicate via an application layered on top of NATS messaging.

- Simulate inter-satellite links using intermittent IP-based connections between satellite programs.

- Demonstrate key operational scenarios, including coordinated imaging tasks, data relay through intermediate satellites, and synchronisation of NATS streams among satellites.

- Measure and evaluate performance indicators such as message delivery reliability, latency, throughput, and stream consistency.

- Analyse the applicability of NATS to real-world satellite constellations, outlining both benefits and limitations.

2 Context and State-of-the-Art

2.1 Satellite Constellations and the Role of ISLs

The concept of satellite constellations has evolved significantly over the past three decades. Initially dominated by large-scale, government-funded systems such as Iridium and Globalstar, the landscape has shifted toward smaller, cost-efficient constellations launched by commercial and academic actors. These distributed systems leverage the economies of scale in small satellite manufacturing and launch, offering increased revisit times, global coverage, and redundancy [1].

A key enabler for the success of such constellations is the inter-satellite link (ISL). ISLs allow satellites to communicate directly with each other without relying solely on ground stations. This reduces latency, increases autonomy, and enables resilience in cases where ground connectivity is intermittent or compromised. ISLs can be implemented using various technologies, including radio frequency (RF), optical (laser), and emerging visible-light communication systems [2]. RF-based ISLs are mature and widely deployed, but optical ISLs promise higher bandwidth and lower interference at the expense of pointing accuracy and susceptibility to atmospheric effects.

In Low Earth Orbit (LEO) constellations, ISLs are particularly valuable due to the dynamic topology of the network. Satellites in LEO move rapidly relative to the Earth's surface, and the window for ground contact at any given station is limited.

Inter-satellite networking mitigates this by enabling information to propagate across the constellation until a downlink opportunity arises. Systems like SpaceX Starlink have already demonstrated operational deployment of ISLs to provide broadband internet services [3]. In scientific constellations, ISLs enable domain data sharing which enhances mission performance by fusing multiple modalities.

2.2 Communication Protocols and Network Layer Considerations

A recurring challenge in inter-satellite networking is the design of communication protocols that are resilient to intermittent connectivity and variable latency. Traditional TCP/IP was not designed for space, where links may frequently break and reestablish due to orbital dynamics. Nonetheless, IP-based networking is attractive because of its ubiquity, mature tooling, and compatibility with ground-based systems. Research efforts have therefore sought to adapt IP networking to space contexts by incorporating delay-tolerant networking (DTN) principles and opportunistic routing strategies [4].

Yin *et al.* [3] propose INTCP, a TCP variant tailored for inter-satellite communication, incorporating information-centric principles and hop-by-hop retransmissions. Their findings suggest significant performance improvements in latency and jitter over classical TCP in LEO environments. Another recent approach, LeoTCP, also addresses low-latency and high-throughput requirements in dynamic LEO environments [5]. While DTN and these enhanced transport protocols improve reliability, there is still a need for lightweight, application-layer messaging protocols that can operate under constrained satellite resources.

2.3 Synthetic Aperture Radar (SAR) and Multi-Modal Constellations

Synthetic Aperture Radar (SAR) is an advanced remote sensing technology that actively emits microwave signals and measures the backscatter from the Earth's surface. Unlike optical sensors, SAR operates independently of daylight and is largely unaffected by cloud cover, weather conditions, or seasonal illumination variations. This characteristic makes SAR particularly valuable for continuous Earth observation and applications requiring reliable, all-weather imaging [6].

2.3.1 Capabilities and Applications of SAR

SAR satellites can capture high-resolution images over wide swathes of terrain. Key capabilities include:

- **High-Resolution Imaging:** Modern SAR systems, such as the TanDEM-X formation, achieve resolutions down to a few meters, enabling detailed topographic mapping and infrastructure monitoring [6].
- **Change Detection:** By comparing repeated acquisitions, SAR allows for accurate detection of changes in land cover, deforestation, urban expansion, or post-disaster damage assessment [1].
- **Interferometry:** SAR interferometry (InSAR) measures phase differences between acquisitions to determine surface displacement, topography, and deformation over time [6].

The large volumes of data generated - often several GBs per pass - necessitate efficient on-board processing, task scheduling, and data relaying mechanisms to maximize the value of each acquisition [5].

2.3.2 Multi-Modal Persistent Monitoring

To enhance situational awareness and decision-making, modern radar constellations leverage the unique physics of Synthetic Aperture Radar (SAR) to overcome the limitations of traditional orbital sensing. Key advantages of a dedicated SAR-based approach include:

- **Atmospheric Independence:** Unlike passive sensors, SAR provides high-resolution imaging through dense cloud cover, smoke, and haze, ensuring consistent data acquisition regardless of meteorological conditions [1].
- **Illumination Independence:** By utilizing active microwave sensing, SAR satellites operate effectively during the eclipse portion of an orbit, providing a true 24-hour monitoring capability that is not restricted by solar illumination [6].
- **Phase and Polarimetric Diversity:** SAR offers multi-dimensional data through varying polarizations (e.g., HH, VV, VH) and interferometric phases, enabling the detection of surface changes, material properties, and fine-scale movements that are invisible to the human eye [1].

The synergy of these radar-specific capabilities allows for persistent situational awareness. For example, a wide-swath SAR scan can identify broad changes in a maritime environment, which can automatically trigger a high-resolution "Spotlight" SAR acquisition. Such intra-modal tasking ensures that observation latency is minimized while maintaining a high level of detail across dynamic operational theaters [1].

2.3.3 Coordination Challenges in Multi-Modal Constellations

Multi-modal constellations introduce several challenges for coordination:

- **Tasking Latency:** Rapidly assigning imaging or sensing tasks across satellites requires low-latency inter-satellite communication [3].
- **Data Prioritization:** Limited downlink capacity necessitates intelligent prioritization of acquired data to ensure the most critical information reaches ground stations first [5].
- **Distributed Decision-Making:** Satellites must autonomously coordinate actions based on observations from peers, particularly when some nodes lack real-time ground connectivity [7].
- **Synchronization:** Multi-sensor acquisitions must be temporally aligned to enable accurate data fusion and analysis [1].

Efficient middleware and messaging frameworks, such as NATS with JetStream, are therefore critical for enabling these coordination tasks. They provide mechanisms for reliable, asynchronous message delivery, allowing satellites to exchange tasking commands, metadata, and telemetry even under intermittent connectivity [8], [9].

2.3.4 Operational Benefits of Multi-Modal Coordination

By integrating multi-satellite capabilities in a cooperative constellation, the following operational benefits can be achieved:

- **Improved Situational Awareness:** Fusing complementary data streams enhances the detection, verification, and characterization of events on the ground [1].
- **Increased Data Utility:** Coordinated observations reduce redundancy, optimize sensor utilization, and enable timely delivery of actionable information [5].

- **Autonomous Decision Support:** Distributed satellites can make coordinated decisions on task assignments and data relays, reducing reliance on ground stations for real-time control [7].

- **Resilience to Ground Segment Limitations:** Satellites without immediate ground connectivity can leverage peers to relay data or receive commands, maintaining operational continuity [3].

2.3.5 SAR Data Considerations for Middleware Design

The high data volume generated by SAR satellites places stringent requirements on middleware design:

- **Throughput Optimization:** Messaging systems must efficiently handle large object transfers between satellites or to ground stations [5].

- **Reliability under Intermittent Connectivity:** Mechanisms such as message persistence and replay ensure that critical SAR data is not lost during brief network disruptions [9].

- **Scalability:** Middleware should scale to support multiple satellites exchanging high-resolution imagery simultaneously without excessive latency [7].

2.3.6 Summary

Multi-modal constellation provides significant advantages for continuous, high-fidelity Earth observation. Coordinating these satellites requires low-latency, reliable, and scalable middleware to manage tasking, data distribution, and autonomous decision-making. The challenges posed by high data volumes, intermittent connectivity, and distributed operations make middleware design a critical aspect of realizing the full potential of multi-modal satellite constellations [1], [3], [5]–[7].

2.4 Middleware and Messaging Paradigms

Middleware forms a foundational layer in distributed systems by decoupling application logic from the complexities of underlying network communication. By providing standardized abstractions for message exchange, middleware enables scalability, fault tolerance, and modular system design across heterogeneous environments. In terrestrial distributed systems and Internet of Things (IoT) deployments, middleware has been extensively studied as a means of managing asynchronous communication, service discovery, and state propagation under varying network conditions [7].

Traditional enterprise-oriented messaging protocols such as the Advanced Message Queuing Protocol (AMQP) emphasize transactional reliability, broker-mediated delivery, and strict delivery guarantees. These characteristics make AMQP suitable for data integrity–critical applications but also introduce non-trivial protocol overhead and operational complexity. In contrast, Message Queuing Telemetry Transport (MQTT) was designed specifically for constrained environments, prioritizing minimal bandwidth usage and lightweight publish–subscribe semantics. MQTT has therefore seen widespread adoption in IoT systems, where devices often operate under limited power, compute, and connectivity constraints [10].

Another important class of middleware is represented by real-time data-centric frameworks such as the Data Distribution Service (DDS). DDS provides fine-grained Quality of Service (QoS) controls over latency, reliability, and delivery deadlines, making it well suited for mission-critical and real-time systems, including avionics and defense applications. However, these guarantees come at the cost of increased configuration complexity and a comparatively heavy runtime footprint, which can limit applicability in resource-constrained satellite platforms [11].

Distributed log-based systems such as Apache Kafka represent a different design philosophy, focusing on high-throughput, durable event streaming and strong

persistence guarantees. Kafka has become a dominant middleware solution in large-scale data processing and analytics pipelines due to its ability to handle massive data volumes with replayable event histories. Nevertheless, Kafka's reliance on cluster coordination, persistent storage, and relatively heavyweight operational requirements makes it poorly suited for deployment in environments with intermittent connectivity and strict Size, Weight, and Power (SWaP) constraints, such as small satellite constellations [12].

Across these paradigms, a consistent trade-off emerges between reliability, scalability, latency, and resource consumption. While terrestrial systems often assume stable connectivity and abundant computational resources, space systems operate under fundamentally different assumptions. Satellites must tolerate frequent link disruptions, variable latency, and prolonged isolation from both ground infrastructure and peer nodes. As a result, middleware designed for space applications must emphasize low protocol overhead, resilience to intermittent connectivity, and decentralized operation rather than strict consistency or centralized coordination [1].

These constraints motivate increasing interest in lightweight, message-oriented middleware that can operate effectively at the application layer while remaining agnostic to the underlying transport and physical communication medium. Such systems align naturally with the principles of software-defined space systems, where adaptability, autonomy, and graceful degradation are prioritized over rigid network assumptions. Within this context, evaluating modern lightweight messaging frameworks - originally developed for cloud, edge, and IoT environments - as candidates for inter-satellite communication becomes a relevant and timely research direction.

2.5 NATS Messaging System

NATS is an open-source, lightweight messaging system designed to support high performance, scalability, and simplicity in distributed environments [8]. It provides

a compact core messaging infrastructure that supports multiple communication patterns commonly required in distributed systems, allowing applications to select the interaction model that best fits their coordination needs. These patterns include publish–subscribe communication for one-to-many dissemination of events, request–reply interactions for synchronous exchanges where a response is expected, and queue groups that enable multiple subscribers to cooperatively process workloads while ensuring that each message is handled by only one participant.

The publish–subscribe paradigm is particularly well suited for scenarios where information must be disseminated broadly and efficiently, such as broadcasting satellite tasking commands, orbital state updates, or constellation-wide coordination signals. In this model, publishers and subscribers remain loosely coupled, which allows nodes to dynamically join or leave the system without requiring reconfiguration of peer endpoints. Request–reply communication, by contrast, enables more tightly coordinated interactions where one node explicitly solicits information or action from another, for example when querying resource availability, confirming task execution, or negotiating scheduling decisions. Queue groups extend these patterns by enabling load sharing across multiple subscribers, allowing computational or communication tasks to be distributed while maintaining simple semantics at the messaging layer.

Together, these communication patterns allow NATS to support both event-driven and coordination-oriented workflows within a single system. This flexibility is particularly valuable in distributed satellite systems, where communication requirements can vary significantly depending on mission phase, operational context, and network conditions.

2.5.1 JetStream Persistence and Streaming

JetStream is an extension of the core NATS system that introduces message persistence, stream abstraction, and stronger delivery guarantees [9]. By enabling mes-

sages to be retained either in memory or on disk, JetStream allows data to survive beyond the lifetime of individual connections, supporting message replay and recovery after disconnection. This capability is especially important in space environments, where intermittent connectivity is not an exception but a fundamental characteristic of system operation.

Through the concept of streams, JetStream organizes messages into durable sequences that can be consumed by one or more subscribers. Each subscriber, or consumer, maintains its own independent consumption state, allowing multiple processing workflows to operate on the same data without interference. This design enables flexible data dissemination patterns, such as one satellite consuming real-time updates while another processes historical data for delayed analysis.

Reliability is further enhanced through acknowledgment-based delivery semantics. Subscribers explicitly acknowledge messages once they have been successfully processed, allowing the system to detect failures and automatically redeliver unacknowledged messages. This mechanism provides at-least-once delivery guarantees, ensuring that critical commands or data products are not silently lost during transient failures. In the context of inter-satellite communication, such guarantees are essential for maintaining consistent operational state across nodes despite frequent link disruptions and variable latency.

2.5.2 Lightweight Design and Performance Characteristics

A defining feature of NATS is its minimal protocol overhead, which directly contributes to its suitability for deployment on constrained satellite hardware. The core NATS server is intentionally designed to be compact and efficient, reducing memory footprint and simplifying operational management. Despite this simplicity, the system is capable of handling very high message throughput with low end-to-end latency in terrestrial environments, demonstrating that efficiency does not necessarily

require architectural complexity.

Low latency message delivery is maintained even under high load conditions, which is important for time-sensitive coordination tasks such as synchronized observations or rapid task reassignment. At the same time, NATS places a strong emphasis on resilience at the client level. Clients are designed to automatically reconnect after network disruptions and to resume message flow without requiring extensive application-level recovery logic. This behavior allows distributed nodes to tolerate transient failures gracefully, an essential property in networks where connectivity can be lost and restored frequently.

From a scalability perspective, NATS supports horizontal scaling through clustering of multiple servers. This allows the messaging infrastructure to grow incrementally as the size of a satellite constellation increases, without requiring fundamental changes to the communication model. Such scalability is particularly relevant for modern space systems, where constellations may evolve over time and incorporate satellites with heterogeneous capabilities.

2.5.3 Applicability to Inter-Satellite Communication

The combination of lightweight operation, flexible messaging patterns, and optional persistence makes NATS particularly attractive for inter-satellite communication scenarios. Publish–subscribe and request–reply interactions enable distributed satellites to coordinate observation schedules, exchange status information, relay signals, or collaboratively plan imaging tasks in a decentralized manner. Queue groups further allow processing workloads to be distributed across multiple nodes, supporting efficient utilization of onboard computational resources.

JetStream persistence plays a crucial role in ensuring robustness under intermittent connectivity. When a satellite temporarily loses contact with its peers or the ground segment, critical messages can be retained and delivered once connectivity

is restored. This capability supports reliable command dissemination and data relay without requiring continuous end-to-end links, which are often infeasible in Low Earth Orbit environments.

NATS also aligns naturally with the dynamic topology of inter-satellite networks. As satellites move relative to one another, communication links form and dissolve continuously. NATS clients automatically handle reconnection and can resume message consumption without explicit resynchronization logic at the application level. This behavior reduces system complexity and supports the inherently time-varying connectivity patterns of satellite constellations.

Community experience has demonstrated that NATS performs reliably in edge and IoT deployments characterized by constrained bandwidth, limited compute resources, and intermittent connectivity [13]. While space systems impose additional constraints and operational challenges, the similarities in network behavior suggest that the underlying design principles of NATS can be effectively extended to inter-satellite communication contexts.

2.5.4 Limitations and Design Trade-offs

Despite its advantages, NATS is not without limitations when considered for space applications. The core NATS protocol prioritizes availability and low latency over strict consistency, offering best-effort, at-most-once delivery semantics in its simplest configuration. While JetStream mitigates this limitation by providing persistence and acknowledgment-based delivery, these features introduce additional resource overhead and system complexity that must be carefully managed on constrained platforms.

Furthermore, NATS operates at the application layer and assumes the availability of an underlying IP-based transport. As a result, it does not address challenges at the physical or link layers, such as link scheduling, beam steering, or modulation

schemes, which remain the responsibility of lower-level communication subsystems. However, this separation of concerns can also be viewed as a strength, as it allows NATS to be integrated alongside existing flight software stacks and communication protocols without requiring invasive architectural changes.

These trade-offs position NATS not as a replacement for traditional space communication frameworks, but as a complementary middleware layer that enables flexible, software-defined coordination mechanisms. Understanding and evaluating these trade-offs in the context of inter-satellite communication forms a central motivation for the experimental investigations presented in later chapters.

2.5.5 Summary

Overall, NATS, combined with JetStream, offers a lightweight, resilient, and scalable messaging middleware capable of supporting distributed coordination in satellite constellations. Its low overhead, support for multiple messaging paradigms, and persistence mechanisms provide a strong foundation for building software-defined space systems that require asynchronous, reliable communication across dynamically connected nodes.

2.6 Middleware in Space Systems

Middleware has long played an important role in space systems by providing reusable abstractions for onboard software management, communication, and mission operations. As satellite missions have grown more complex and software-driven, both space agencies and research communities have increasingly adopted modular middleware frameworks to improve portability, maintainability, and operational flexibility. These frameworks aim to reduce mission-specific development effort by offering standardized services for application lifecycle management, telemetry handling, and

command execution.

The European Space Agency's NanoSat Mission Operations Framework (NMF) exemplifies this trend toward service-oriented middleware in space systems. NMF provides a modular, application-based architecture that supports the deployment, monitoring, and updating of software components on nanosatellites during mission operations [14], [15]. By treating onboard functionality as a collection of services, NMF enables greater flexibility in mission configuration and allows software updates to be applied after launch. This approach reflects a shift away from monolithic flight software toward more adaptable and evolvable system designs, particularly for small satellite platforms.

Similarly, NASA's Core Flight System (cFS) has established a reusable and modular software framework that has been widely adopted across multiple missions [16]. cFS provides a layered architecture consisting of a core flight executive and a set of reusable services that support inter-process communication, time management, event reporting, and application scheduling. By separating mission-independent infrastructure from mission-specific applications, cFS enables portability across different hardware platforms and operating systems, significantly reducing development time and risk for new missions [17].

While both NMF and cFS demonstrate the clear benefits of middleware abstractions in space systems, their primary focus lies in ground-to-satellite interaction and onboard application management. These frameworks are designed to support command and control, telemetry processing, and software lifecycle management within a single spacecraft or between a spacecraft and the ground segment. As such, they assume relatively stable communication patterns and centralized operational authority, which align well with traditional mission concepts.

In contrast, emerging satellite constellations increasingly require direct satellite-to-satellite coordination to support distributed sensing, cooperative task execution,

and autonomous decision-making. In such systems, communication patterns are inherently dynamic, and centralized ground-based coordination may be impractical due to latency, bandwidth constraints, or limited contact opportunities. Existing space middleware frameworks provide limited native support for decentralized, peer-to-peer coordination across multiple spacecraft, highlighting a gap between established flight software architectures and the needs of modern distributed space systems [1], [18].

This gap has motivated growing research interest in software-defined and autonomous space systems, where higher-level middleware layers enable flexible coordination without tightly coupling applications to underlying communication technologies. Rather than replacing established flight software frameworks, such middleware can complement them by operating at the application layer, enabling satellites to exchange state information, coordinate tasks, and adapt behavior in response to changing network conditions. Within this context, lightweight messaging systems originally developed for cloud and edge environments offer a promising foundation for extending middleware concepts beyond individual spacecraft toward constellation-wide coordination.

2.7 Existing Research Gaps

While inter-satellite communication (ISC) has seen significant theoretical progress [1], a critical gap remains between high-level network simulations and the practical requirements of software-defined, autonomous space systems. The following sections detail the primary areas where current research remains insufficient for operational deployment in modern small satellite constellations.

2.7.1 Application-Layer Middleware for Constellation Autonomy

A primary research gap exists in the adaptation of cloud-native, message-oriented middleware (MOM) to the unique constraints of the orbital environment. Most current space-qualified software frameworks, such as the European Space Agency's NanoSat Mission Operations Framework (NMF) [14] or NASA's Core Flight System (cFS) [16], prioritize ground-to-satellite management or internal on-board application isolation rather than direct satellite-to-satellite coordination. While terrestrial studies have compared messaging systems like Kafka and NATS [7], there is a notable lack of systematic analysis regarding how lightweight, publish-subscribe paradigms can handle high topological change frequency and limited Size, Weight, and Power (SWaP) budgets [2]. While terrestrial cloud computing assumes plentiful bandwidth, satellites act as heterogeneous nodes that must independently manage intermittent IP-based connectivity without relying on continuous ground station visibility [13].

2.7.2 Coordination of Multi-Modal Heterogeneous Constellations

Current research frequently investigates uniform constellations composed of identical sensors, but significant gaps remain in the collaborative tasking of multi-modal systems [6]. Although the synergy between these modalities is recognized as a key enabler for enhanced data fusion, the middleware architectures required to trigger autonomous cross-domain tasking are largely unexplored [18]. There is a lack of documented solutions that address the specific coordination challenges of multi-sensor acquisitions that must be temporally aligned and prioritized under limited downlink capacity [4].

2.7.3 Scarcity of Hardware-in-the-Loop and Flatsat Validation

A substantial portion of existing ISC literature relies heavily on mathematical simulations or high-level network models, with fewer laboratory-based demonstrations using flatsats or hardware-in-the-loop setups. There is a critical need for experimental evidence using process-level isolation to replicate the realistic failure domains of independent physical satellites. Without such environments, subtle distributed system anomalies - such as protocol-level routing loops in symmetric peer-to-peer (P2P) topologies or redundant task re-broadcasts - often remain undetectable in purely theoretical models. Current research often overlooks the practical implementation details of how separate binaries on isolated hardware handle state synchronization during the brief, sporadic windows of connectivity typical in Low Earth Orbit (LEO).

2.7.4 Resilience and Recovery in Distributed Space Systems

Existing studies often prioritize network throughput and latency, yet there is a significant gap in addressing systemic resilience and state recovery following satellite crashes or power cycles in orbit. While traditional transport protocols like INTCP [3] and LeoTCP [5] have been optimized for the satellite medium, they do not resolve the higher-layer problem of persisting pending tasks across intermittent connection losses. Research into integrating local persistence mechanisms, such as NATS JetStream [9], within federated gateway architectures is sparse. Specifically, there is an absence of design principles that allow for durable, sender-responsible store-and-forward models that survive process termination without the complexity of terrestrial clustering, which has shown vulnerabilities in partition tolerance [19].

2.7.5 Scalability and Loop-Free Routing in Symmetric Topologies

As constellations evolve into larger scales, the transition from centralized broker models to true peer-to-peer mesh federation remains a significant bottleneck [8]. Most current systems still utilize ground-centric control or centralized in-orbit brokers, which introduce significant latency and single points of failure. There is a lack of research focusing on loop-free routing for symmetric satellite federation, particularly when using leaf-node protocols that assume a hierarchical relationship. Systematic analysis of how decentralized architectures can maintain eventual task convergence across N-satellite topologies while balancing delivery robustness against limited onboard resource consumption is essential for the next generation of software-defined space systems.

2.7.6 Synthesis of Research Contributions

To address the research gaps identified in the preceding sections, this thesis proposes a series of six experiments. Table 2.1 provides a synthesized overview of how the experimental methodology aligns with the limitations identified in current inter-satellite communication literature.

2.8 Key Definitions

This section provides a standardized vocabulary for the technical concepts discussed throughout this thesis, spanning satellite engineering, distributed systems, and the specific messaging protocols utilized in the experimental phases.

Inter-Satellite Link (ISL): A communication channel directly connecting satellites in orbit, implemented via RF, optical, or hybrid technologies.

Table 2.1: Mapping Research Gaps to Experimental Solutions

Research Gap	**Exp.**	**Experimental Contribution**
Application-Layer Middleware for Autonomy	1	Validated NATS as a lightweight broker for satellite-to-satellite messaging.
	2	Shifted to NATS Leaf Nodes to remove centralized dependencies.
Coordination of Multi-Modal Constellations	1	Implemented a priority-based protocol for multi-modal constellation.
	3	Validated multi-hop task routing across different satellite identities.
Scarcity of Formal HITL Validation	3	Transitioned to isolated binary executables to simulate physical process separation.
	5	Validated scalability on a simulation of up to 5 independent satellite nodes.
Resilience and State Recovery	4	Introduced sender-responsible store-and-forward to handle intermittent links.
	6	Integrated local JetStream persistence to survive satellite crashes and power cycles.
Scalability and Loop-Free Routing	4	Replaced Leaf Nodes with NATS Gateways to enable loop-free symmetric P2P topologies.
	5	Evaluated connection complexity and message deduplication in N-satellite full mesh.

Middleware: A software layer providing abstractions to simplify and standardize communication between distributed applications.

Message-Oriented Middleware (MOM): Middleware designed around messaging paradigms such as publish/subscribe and message queues.

NATS JetStream: A persistence and streaming extension to NATS enabling reliable, asynchronous communication with at-least-once guarantees.

Flatsat: A laboratory hardware setup replicating satellite components and interfaces for software development and validation.

Software-Defined Space Systems: Space systems whose functionality and behavior are predominantly defined by adaptable software layers rather than fixed hardware.

NATS Leaf Node: A NATS protocol used to create a hierarchical topology where a local server transparently routes messages and propagates subscriptions to a central or upstream server.

NATS Gateway: A peer-to-peer federation mechanism explicitly designed for symmetric, loop-free connectivity between independent NATS clusters or servers.

Multi-Modal Constellation: A satellite constellation integrating diverse sensing technologies to enhance situational awareness through collaborative tasking.

Store-and-Forward: A communication technique where data is sent to an intermediate node (or retained locally) and kept until a valid link to the destination becomes available.

Sender-Responsible Delivery: An application-layer logic where the originating satellite retains responsibility for task delivery, retrying the transmission until success is confirmed via state synchronization.

Message Deduplication: The process of ensuring that a message reaching a destination through multiple paths is processed only once, typically managed via unique message identifiers.

Work Queue Policy: A NATS JetStream retention policy where messages are removed from a stream only after a consumer acknowledges successful processing, ensuring no data is lost during intermittent connectivity.

Routing Loop Anomaly: A protocol-level error occurring in symmetric leaf node topologies where NATS detects circular connection paths and rejects the link to prevent infinite message circulation.

Origin-Based Filtering: A synchronization logic where satellites maintain distinct collections for local and received tasks, ensuring that tasks are only broadcast by their original creator to prevent redundant propagation.

Process-Level Isolation: A simulation method where independent satellite entities run as separate binary executables with isolated memory spaces and failure domains to approximate physical separation in orbit.

State Introspection: A mechanism using request/reply messaging to programmatically query the internal status of a simulated satellite, including its known peers and task collections.

Symmetric Topology: A network configuration where satellites are treated as equal peers without a fixed hierarchy, allowing bidirectional links to be established dynamically between any two nodes.

Dual-Server Architecture: A design pattern where each satellite node operates two independent NATS instances: a Gateway server for inter-satellite networking and a standalone JetStream server for local file-based persistence.

Full Mesh Configuration: A network topology where every satellite in the constellation maintains a potential direct connection to every other satellite, increasing redundancy but also increasing connection complexity.

Task Propagation: The process by which a task command or data object is distributed across the constellation through intermediate nodes until it reaches the target satellite or a downlink gateway.

State Convergence: The condition where all satellites in an intermittently connected constellation eventually reach a consistent view of the global task list once network connectivity is restored.

Attempt-Based Heuristic: A cleanup strategy where tasks are removed from a retry queue after a predefined number of delivery attempts or once delivery is inferred through peer state synchronization.

3 Experimentation

Following the identification of critical research gaps in Chapter 2, specifically regarding the lack of decentralized, persistent, and scalable messaging middleware for multi-modal constellations [1], [18], this chapter details the methodology used to address these deficiencies. The primary objective is to evaluate the feasibility and performance of a NATS-based architecture for inter-satellite communication through a systematic, multi-stage experimental approach [8].

3.1 Experimental Roadmap

The experimentation is structured into six distinct phases, as synthesized in Table 2.1. Each experiment progressively addresses architectural or operational challenges identified in the state-of-the-art, following a logical evolution from centralized to decentralized persistent architectures:

- **Experiments 1 & 2**: Establish foundational feasibility by validating NATS as a lightweight broker [7] and removing centralized dependencies through the Leaf Node protocol [14].
- **Experiment 3**: Introduces physical deployment simulation through independent, pre-configured binary executables representing distinct satellite entities.
- **Experiments 4 & 5**: Address scalability and loop-free routing by adopting a Gateway-based federation [8] and validating performance across N-satellite

mesh topologies.

- **Experiment 6**: Finalizes the resilient architecture by integrating local JetStream persistence [9] to ensure state recovery and task durability during node failures, addressing potential partition tolerance limitations [19].

3.2 Setup and Simulation Environment

To ensure reproducibility and realistic simulation, the experiments were conducted using independent binary executables running as independent processes to reflect realistic resource isolation. The system is implemented in Golang - layered on top of NATS messaging. This setup abstracts the physical inter-satellite link layer - such as radio frequency or optical links to isolate and evaluate the messaging infrastructure under simulated intermittent IP-based network conditions where TCP connections are established sporadically.

3.3 Experiment 1: Centralized NATS-Based Inter-Satellite Communication

The initial implementation took the simplest possible approach: hosting a centralized NATS broker on one satellite within the constellation. While this architecture would ultimately prove inadequate for operational deployment, it served a critical purpose in the research progression. By starting with a centralized design, the experiment could validate basic messaging functionality, develop the simulation framework, and establish baseline performance characteristics before introducing the additional complexity of distributed coordination. The limitations that would inevitably emerge from this centralized approach would then provide concrete motivation and design constraints for subsequent experiments.

3.3.1 Objectives

This initial exploration pursued three interconnected goals that would establish the foundation for all subsequent work. First, the experiment needed to assess whether NATS could support inter-satellite message routing in a resource-constrained environment where processing power, memory, and bandwidth are limited compared to terrestrial systems. This assessment would determine if the fundamental messaging primitives - publish-subscribe patterns, subject-based routing, and message persistence - could operate reliably under satellite-like constraints.

Second, a communication protocol specifically tailored for satellite coordination and data exchange needed to be designed and implemented. Unlike general-purpose messaging protocols, satellite communication requires explicit handling of message priorities to distinguish between routine telemetry and time-critical emergency responses. The protocol would need to support multiple operational modes including periodic health monitoring through telemetry broadcasts, dynamic task assignment for coordinating satellite activities, relay operations for data forwarding across the constellation, and system control messages for managing satellite behavior during anomalous conditions.

Third, building a realistic simulator to evaluate satellite network interactions under controlled scenarios would provide the experimental testbed necessary for systematic investigation. This simulator would need to approximate the isolation characteristics of physical satellites, enabling discovery of failure modes that might not manifest in simpler test environments.

3.3.2 System Architecture

The experimental constellation consisted of multiple simulated satellites, each executing as an independent operating system process to reflect the resource isolation inherent in physical satellite deployments. This process-level separation was not

merely a convenience - it ensured that memory corruption, resource exhaustion, or crashes in one satellite would not cascade to others, mirroring the fault isolation properties that actual satellites possess when operating in the harsh environment of space. Each process maintained its own address space, file descriptors, and execution context, communicating with peer satellites only through well-defined message interfaces.

Within this multi-process architecture, one satellite was designated as the central NATS broker, hosting the messaging infrastructure that would enable inter-satellite coordination. All other satellites connected to this central node as clients, publishing messages to specific subjects and subscribing to receive messages from topics relevant to their operational responsibilities. Communication between satellites occurred indirectly through this central broker, which managed message routing, maintained subscription information, and ensured that published messages reached all interested subscribers. This centralized architecture represented a natural starting point for the investigation - it minimized implementation complexity by consolidating routing logic in a single location while allowing validation of core messaging functionality before introducing the additional complexities of distributed coordination.

The messaging protocol organized communication through a hierarchical subject namespace that reflected the operational semantics of satellite coordination. Telemetry data, representing the routine health and status information that satellites continuously generate, was published to the `satellite.telemetry.*` subject hierarchy, where the wildcard allowed individual satellites to publish to satellite-specific topics like `satellite.telemetry.SATELLITE-ALPHA` or `satellite.telemetry.SATELLITE-BETA`. Task assignments, which represented commands for satellites to perform specific activities such as capturing imagery over designated targets or relaying data through specific communication paths, were distributed through the `satellite.task.*` subject space. Data relay operations,

essential for forwarding collected information across the constellation when direct ground contact was unavailable, utilized the `satellite.relay.*` subjects. System control messages, reserved for managing satellite behavior during anomalous conditions or coordinating constellation-wide reconfigurations, were published to `satellite.control.*`. This subject-based organization enabled selective message filtering and subscription management, allowing each satellite to receive only the messages relevant to its current operational state [Figure 3.1].

The architecture leveraged NATS JetStream to provide message persistence and replay capabilities, addressing one of the fundamental challenges of satellite communication: the intermittent nature of connectivity. When a satellite temporarily loses contact with the constellation - whether due to orbital geometry, antenna pointing constraints, or onboard system failures - messages published during the disconnection period must not be lost. JetStream's persistent streams ensured that messages would survive process restarts and could be recovered after temporary failures. Each message type was assigned to a dedicated JetStream stream. Telemetry streams, representing high-frequency routine data, employed size-based limits to prevent unbounded growth while retaining recent history. Task assignment streams utilized work-queue semantics where messages were removed only after explicit acknowledgment of completion. This persistence layer transformed the ephemeral publish-subscribe model into a durable messaging system capable of surviving the transient failures characteristic of space operations.

An application was developed to provide structured communication between satellites, moving beyond simple text-based messages to rich, typed data structures that captured the metadata essential for reliable coordination. Each message carried a unique identifier generated using a combination of the originating satellite's identity and a monotonically increasing sequence number, enabling precise tracking of message flow through the system and detection of duplicate deliveries.

To enable interactive experimentation and validation, a mission control interface was developed in the form of a command-line tool that operators could use to inject tasks, modify network conditions, and observe constellation behavior. This Task CLI supported real-time task assignments to specific satellites, allowing us to inject imaging requests, relay operations, or maintenance activities and observe how the constellation responded. Dynamic control of network link states enabled simulation of communication disruptions, orbital geometry changes, and equipment failures by programmatically enabling or disabling connectivity between specific satellite pairs. Satellite isolation simulations allowed testing of store-and-forward behavior by completely disconnecting individual satellites from the constellation and later reconnecting them to observe message synchronization. This interactive interface proved invaluable during development, enabling rapid hypothesis testing and providing immediate feedback on system behavior that would have been difficult to obtain through batch-mode testing alone.

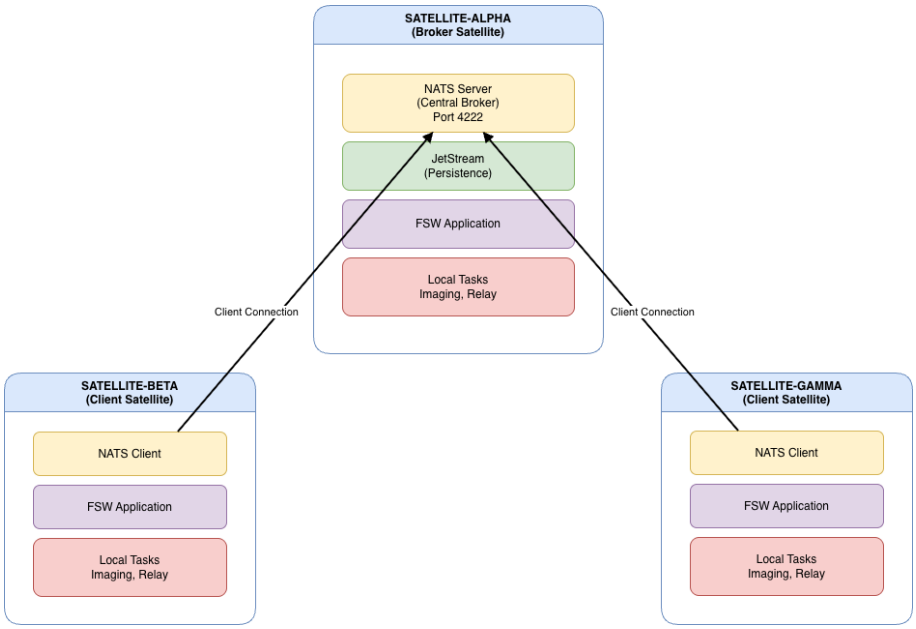

Figure 3.1: Centralized NATS-based Inter-satellite Communication

3.3.3 Key Findings

Functional inter-satellite communication was achieved through the NATS broker hosted on a designated satellite, with all routing handled transparently by the NATS infrastructure. When SATELLITE-ALPHA published an imaging task to the `satellite.task.SATELLITE-BETA` subject, SATELLITE-BETA reliably received the message and could respond with acknowledgment. The multi-hop nature of this communication - with messages traversing from the publishing satellite's process through the broker process to the subscribing satellite's process - introduced minimal latency, typically under 10 milliseconds on the development system, suggesting that message-oriented middleware would not introduce prohibitive overhead even in more resource-constrained environments.

Message persistence and replay via JetStream proved essential for handling the intermittent connectivity scenarios that characterize satellite operations. During controlled experiments where satellites were deliberately crashed and restarted, pending tasks persisted in JetStream streams were successfully recovered and processed after restoration, with no message loss observed. When SATELLITE-BETA was taken offline while SATELLITE-ALPHA published three imaging tasks, subsequent reconnection resulted in all three tasks being delivered in their original order, demonstrating that the system could tolerate temporary partitions without losing operational directives.

The multi-satellite process simulation provided confidence that the experimental framework could reveal behaviors relevant to actual deployments. Satellites operated concurrently with overlapping message exchanges, creating race conditions and timing dependencies similar to those that would exist in physical constellations. Observation of these interactions revealed subtle ordering dependencies - for example, task acknowledgments occasionally arrived before the associated task completion notifications due to variable message processing latencies. These observations, while

minor in this controlled environment, foreshadowed the need for explicit sequence tracking and causal ordering mechanisms in more complex distributed scenarios.

The reusable satellite simulation framework developed during this phase - including the multi-process architecture, message protocol definitions, and interactive testing tools - would serve as the foundation for all subsequent experiments, reducing implementation effort and ensuring consistency across the experimental progression.

3.3.4 Limitations

The most critical limitation was the single point of failure introduced by hosting the NATS broker on one designated satellite. All communication depended on this central node remaining operational - if the broker satellite experienced a power failure, processor crash, or any other fault that terminated the NATS server process, the entire constellation immediately lost connectivity. Experiments confirmed this vulnerability: when the broker process was deliberately killed, all satellite-to-satellite communication ceased within seconds, and satellites could not exchange messages until the broker was restarted.

The centralization bottleneck manifested not only as a reliability concern but also as a scalability constraint. Every message published anywhere in the constellation had to traverse the broker satellite, creating a concentration point for both computational load and network bandwidth. As message rates increased during scaling experiments - simulating scenarios where multiple satellites were simultaneously performing high-frequency telemetry updates and coordinating complex multi-satellite imaging campaigns - the broker satellite's CPU utilization climbed proportionally. While the resource consumption remained manageable for the three-satellite topology tested, extrapolation suggested that constellations with tens or hundreds of satellites would quickly overwhelm a single broker node. Profiling data showed that message routing consumed up to 40% of the broker satellite's CPU time during high-

throughput experiments, leaving insufficient headroom for the satellite to perform its own operational tasks such as sensor management or data processing [Figure 3.2].

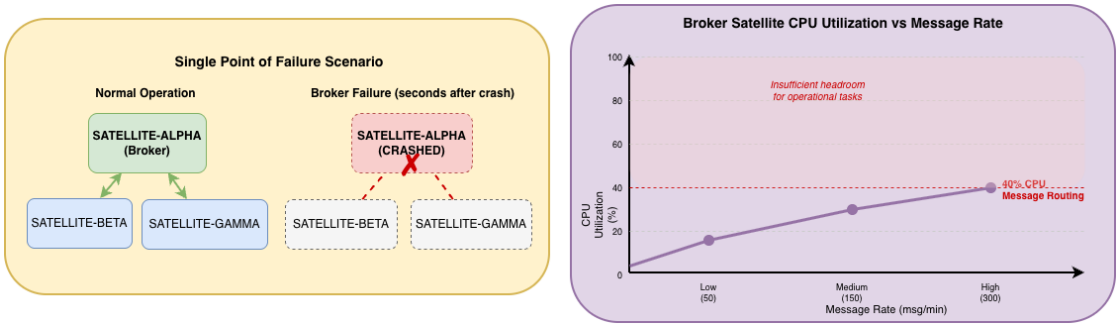

Figure 3.2: Centralization Limitations and Performance Bottlenecks

3.3.5 Outcome and Next Steps

Experiment 1 served as a foundational step in developing and validating the communication and simulation framework that would underpin all subsequent research. The centralized NATS-on-satellite approach successfully enabled initial functionality, proving that message-oriented middleware could operate in a satellite-like environment with process isolation, intermittent connectivity, and resource constraints. The experiment produced three tangible outcomes that would accelerate future work: a working multi-process satellite simulator with realistic fault isolation properties, a structured FSW message protocol supporting priority-based coordination and reliable delivery semantics, and a reusable testing infrastructure with interactive mission control capabilities.

The path forward required investigating distributed and peer-to-peer communication architectures where messaging infrastructure would be spread across multiple satellites rather than concentrated in a single node. Several promising directions emerged from the analysis of Experiment 1's limitations. Decentralized NATS clusters, where multiple satellites could host broker functionality with redundancy and load distribution, might address the single-point-of-failure concern while retaining

NATS's messaging semantics. Alternatively, lightweight brokers distributed across all satellites, each managing a subset of the message routing responsibilities, could eliminate centralization entirely. Direct TCP or UDP-based inter-satellite communication, bypassing middleware abstractions altogether, might reduce overhead and enable satellites to exploit opportunistic communication windows based on orbital geometry. Among these possibilities, NATS Leaf Node protocol emerged as particularly promising.

3.4 Experiment 2: Distributed NATS Leaf Node Architecture

The limitations revealed in Experiment 1 had crystallized a clear imperative: eliminating the single point of failure required distributing the messaging infrastructure across multiple satellites rather than concentrating it in a central broker. The question was no longer whether to distribute, but how to do so while preserving the message persistence, priority handling, and subject-based routing that had proven valuable in the centralized implementation. NATS Leaf Node protocol emerged as a promising architectural approach - it allowed independent NATS servers to form distributed topologies while maintaining transparent message routing between clients connected to different servers. If leaf nodes could provide stable routing semantics under the dynamic connectivity conditions characteristic of satellite operations, they might offer a path toward truly autonomous inter-satellite communication.

This second experiment investigated whether a fully distributed architecture, where each satellite operated its own independent NATS server connected via leaf node relationships, could overcome the centralization bottleneck while maintaining reliable coordination. The architectural shift was fundamental: rather than satellites connecting as clients to a shared broker, each satellite would host its own messaging

infrastructure and establish peer relationships with other satellites' servers. This distribution promised to eliminate the single point of failure, but it also introduced new questions about routing stability, message synchronization across intermittent links, and the coordination required to maintain consistent state when the constellation partitioned and reconnected.

3.4.1 Objectives

The experiment pursued five interconnected goals that addressed the specific limitations identified in Experiment 1. First, eliminating the single point of failure inherent in centralized broker architectures required demonstrating that constellation-wide communication could continue even when individual satellites failed. Unlike the centralized model where broker failure disabled all communication, the distributed architecture needed to maintain connectivity among remaining satellites when any single node went offline.

Second, enabling true peer-to-peer communication between satellites without requiring ground station connectivity would validate that the constellation could operate autonomously during extended periods without ground contact. Satellites needed to exchange messages directly through their leaf node connections rather than routing through intermediary nodes, exploiting opportunistic communication windows created by favorable orbital geometry.

Third, implementing dynamic link control to simulate realistic visibility windows and network partitions would provide the experimental framework necessary to evaluate behavior under operational conditions. Satellite constellations experience constantly changing network topologies as orbital mechanics create and destroy line-of-sight communication paths. The experiment needed to demonstrate that leaf node connections could be established and torn down dynamically to match these changing visibility conditions.

Fourth, validating automatic message synchronization when intermittent connectivity was restored would confirm that satellites could recover from temporary isolation without manual intervention or ground station coordination. When a satellite regained connectivity after a period of isolation, pending messages needed to propagate automatically to newly reachable peers.

Fifth, demonstrating store-and-forward behavior where satellites accumulated tasks locally and synchronized when links became available would validate the delay-tolerant networking properties essential for operational satellite constellations. Satellites often need to buffer operational directives when target satellites are temporarily unreachable, forwarding these messages opportunistically when communication becomes possible.

3.4.2 System Architecture

The architecture represented a fundamental departure from Experiment 1's centralized design. Rather than sharing a single NATS server, each satellite now operated its own independent NATS instance with JetStream enabled for local message persistence. This distribution meant that every satellite could continue processing messages and maintaining local state even when isolated from peers - a critical requirement for autonomous operation. Inter-satellite communication occurred through the NATS Leaf Node protocol, which created routing channels between connected NATS servers without requiring manual configuration of message forwarding rules.

The constellation was configured as a full mesh topology where each satellite could potentially establish leaf node connections to every other satellite. SATELLITE-ALPHA ran its NATS server on port 4222 with the leaf node interface exposed on port 7422. SATELLITE-BETA operated its server on port 4223 with leaf connections accepted on port 7423. SATELLITE-GAMMA hosted its server on port 4224 with the leaf interface on port 7424. This port separation allowed all satellites to

coexist on the development machine while maintaining the process isolation that simulated physical separation.

When a leaf node connection was established between two satellites, NATS automatically propagated subscription information and routed messages transparently based on these subscriptions. The routing mechanism worked through subscription propagation: when SATELLITE-BETA subscribed to `satellite.task.SATELLITE-BETA` on its local server, this subscription information was forwarded through the leaf connection to SATELLITE-ALPHA's server. Subsequently, when SATELLITE-ALPHA published an imaging task to `satellite.task.SATELLITE-BETA`, SATELLITE-ALPHA's server recognized that a remote subscriber existed and forwarded the message through the leaf connection to SATELLITE-BETA's server, which delivered it to the subscribing client. This transparent routing meant that satellite applications could publish and subscribe to messages using their local server without implementing any application-level routing logic [Figure 3.3].

Dynamic connectivity control represented a key innovation in this experiment, addressing the need to simulate the time-varying network topologies that characterize satellite constellations. Three categories of NATS server configurations were developed to represent different connectivity states. Isolated configurations disabled all leaf node connections, leaving the satellite's NATS server operating independently without any remote routing. These configurations simulated periods when a satellite had no line-of-sight communication with any peers - for example, when orbital geometry placed it on the opposite side of Earth from the rest of the constellation. Specific link configurations established a leaf connection to exactly one designated peer satellite, representing scenarios where limited visibility allowed communication with only a subset of the constellation. Full mesh configurations established leaf connections to all other satellites, representing periods of maximum connectivity when

orbital geometry provided simultaneous line-of-sight to all constellation members.

Shell scripts were developed to dynamically restart NATS servers with different configurations, enabling simulation of visibility windows and link restoration events during runtime. When a satellite's connectivity state needed to change - for example, to simulate it emerging from behind Earth and reestablishing communication with the constellation - the controlling script would send a termination signal to the satellite's NATS server process, modify the server's configuration file to include the appropriate leaf node remote definitions, and restart the server. The satellite application would detect the server restart, reconnect automatically, and reestablish its subscriptions, after which messages would begin flowing through the newly established leaf connections. While this restart-based approach introduced brief connectivity interruptions, it provided deterministic control over network topology necessary for systematic experimentation.

The satellite application layer implemented a synchronization protocol with four message types that facilitated distributed coordination. TaskBroadcast messages, published to `satellite.task.*` subjects, announced local tasks to all connected peers. When a satellite created a new imaging task - either through operator command or autonomous planning - it would publish a TaskBroadcast containing the task details, priority level, and origination timestamp. Connected peers receiving this broadcast would store the task locally, enabling them to maintain awareness of constellation-wide activities. TaskAck messages, published to `satellite.ack.*` subjects, acknowledged receipt of tasks from other satellites, providing the originating satellite with confirmation that coordination messages had propagated successfully. Ping and Pong messages, exchanged through `satellite.ping` and `satellite.pong` subjects, enabled connectivity verification and peer discovery. Satellites periodically published Ping messages, and any satellite receiving a Ping would respond with a Pong, allowing the sender to build and maintain a list of currently

reachable peers. SyncRequest and SyncResponse messages, using `satellite.sync.request.*` and `satellite.sync.response.*` subjects, supported full state synchronization between satellites. When a satellite established a new leaf connection, it could request a complete state dump from the newly reachable peer, ensuring that tasks created during periods of isolation would be shared when connectivity was restored.

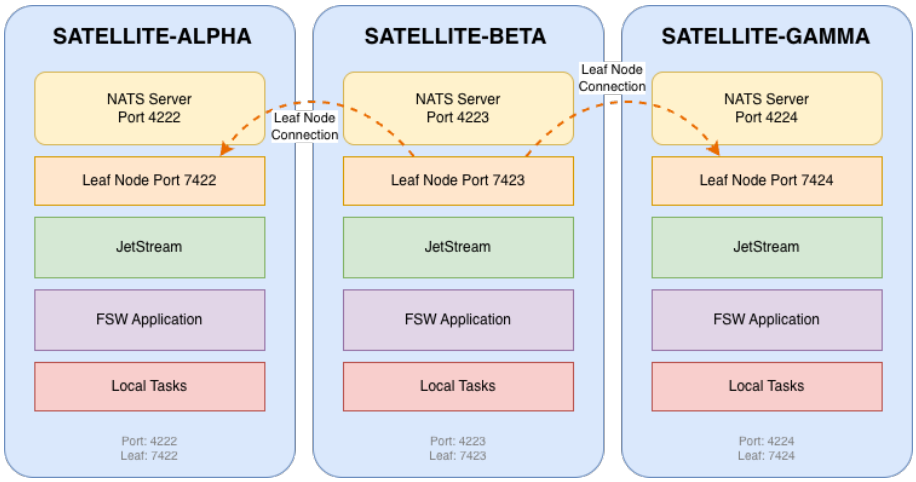

Figure 3.3: Distributed NATS Leaf Node Architecture

3.4.3 Key Findings

The experiment demonstrated that all three satellites could communicate successfully without any central broker or ground station involvement, validating the core hypothesis that distributed NATS infrastructure could support autonomous constellation coordination. When SATELLITE-ALPHA was configured with leaf connections to both SATELLITE-BETA and SATELLITE-GAMMA, imaging tasks published by SATELLITE-ALPHA propagated to both peers within milliseconds, with no central routing point involved in the message delivery path.

The NATS Leaf Node protocol transparently routed messages between connected satellites based on active subscriptions, requiring no application-level routing logic or forwarding rules. The satellite applications simply published to subjects on their

local NATS servers and subscribed to subjects of interest - the leaf node infrastructure handled determining which remote servers needed to receive each published message and establishing the necessary routing paths. This transparent routing significantly simplified application development compared to approaches requiring explicit peer-to-peer connection management or custom routing protocols.

Store-and-forward behavior emerged naturally from the combination of local task storage and leaf node message propagation. During controlled experiments, SATELLITE-ALPHA was started in isolation with two imaging tasks. These tasks were stored locally in SATELLITE-ALPHA's task list but could not propagate because no leaf connections existed. When leaf connections to SATELLITE-BETA and SATELLITE-GAMMA were subsequently enabled by restarting SATELLITE-ALPHA's NATS server with appropriate remote configurations, SATELLITE-ALPHA published TaskBroadcast messages for both tasks. Within seconds, both SATELLITE-BETA and SATELLITE-GAMMA had received and stored copies of SATELLITE-ALPHA's tasks, demonstrating that delayed synchronization functioned correctly across connectivity transitions.

The ping-pong mechanism enabled satellites to discover and track available peers without requiring centralized coordination or global network state. Each satellite maintained a local map of known peers with their last-seen timestamps. When SATELLITE-BETA received a Pong from SATELLITE-GAMMA, it updated SATELLITE-GAMMA's last-seen timestamp and marked it as reachable. If subsequent pings to SATELLITE-GAMMA went unanswered for a configurable timeout period, SATELLITE-BETA would mark SATELLITE-GAMMA as unreachable, allowing the application to adapt its behavior based on current connectivity state.

Selective link control provided precise simulation of visibility windows and network topology changes. Individual satellite-to-satellite links could be enabled or disabled independently by modifying NATS server configurations. During experiments

simulating orbital passes, SATELLITE-GAMMA could be configured with a leaf connection only to SATELLITE-ALPHA, representing a scenario where SATELLITE-BETA was below the horizon from SATELLITE-GAMMA's perspective. Messages published by SATELLITE-GAMMA would propagate only to SATELLITE-ALPHA, accurately reflecting the limited connectivity. Subsequently enabling SATELLITE-GAMMA's leaf connection to SATELLITE-BETA simulated SATELLITE-BETA rising above the horizon, after which messages could flow across all three satellites.

Graceful partition handling demonstrated resilience to partial network failures. When SATELLITE-GAMMA was isolated by removing all its leaf node remotes, SATELLITE-ALPHA and SATELLITE-BETA continued communicating through their active leaf connection. Messages exchanged between SATELLITE-ALPHA and SATELLITE-BETA during SATELLITE-GAMMA's isolation were not lost - when SATELLITE-GAMMA's connectivity was later restored, it could request synchronization and receive updates about activities that occurred during its isolation period.

3.4.4 Limitations

Loop detection overhead occasionally manifested in full mesh configurations where all satellites maintained leaf connections to all peers. NATS leaf nodes are designed with cycle detection to prevent circular routing paths, and in certain timing conditions, this detection would trigger false positives when leaf connections were being established simultaneously from multiple directions. When this occurred, the affected leaf connection would be rejected, requiring a brief delay before automatic reconnection. While these reconnection delays were short - typically 1-2 seconds - they introduced transient communication interruptions during topology changes.

Message deduplication was not provided by the leaf node protocol, meaning

satellites could receive the same message multiple times through different routing paths. In a full mesh topology where SATELLITE-ALPHA had direct leaf connections to both SATELLITE-BETA and SATELLITE-GAMMA, and SATELLITE-BETA also had a direct connection to SATELLITE-GAMMA, a message published by SATELLITE-ALPHA might reach SATELLITE-GAMMA both directly through SATELLITE-ALPHA's leaf connection and indirectly after being forwarded by SATELLITE-BETA. The application layer needed to implement deduplication logic to prevent duplicate task processing, adding complexity that would be addressed in later experiments.

3.4.5 Outcome and Next Steps

Experiment 2 validated that distributed NATS infrastructure using the Leaf Node protocol could enable autonomous inter-satellite communication without centralized dependencies. The architecture successfully demonstrated true peer-to-peer messaging between satellites, where communication occurred directly through leaf connections rather than being routed through intermediate brokers. Resilience to individual satellite isolation was confirmed - when any single satellite went offline, the remaining satellites maintained connectivity and continued coordinating through their active leaf connections. Automatic synchronization upon connectivity restoration worked reliably, with isolated satellites receiving pending tasks when leaf connections were reestablished. Granular control over inter-satellite link topology enabled realistic simulation of orbital visibility windows and network partitions.

The distributed NATS approach represented a significant advancement over the centralized architecture of Experiment 1. By distributing messaging infrastructure across all satellites and using leaf nodes to create routing channels, the experiment eliminated the single point of failure that had compromised the centralized design [Figure 3.4]. Each satellite could now continue operating with local message persis-

tence even when isolated from peers, and the constellation could tolerate individual satellite failures without losing overall communication capability.

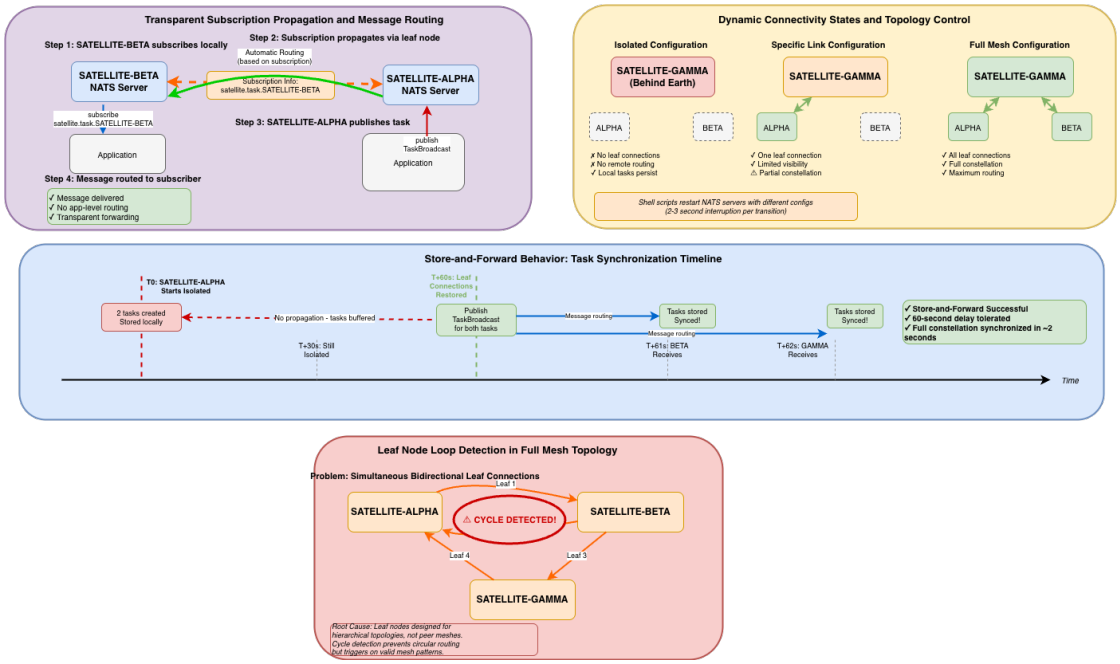

Figure 3.4: Leaf Node Protocol Analysis and Connectivity Patterns

However, the experiment also revealed several areas requiring further investigation. Message deduplication would be essential for preventing duplicate task processing in multi-path topologies where messages could reach destinations through multiple routing paths. Priority-based routing might allow the leaf node topology to consider message urgency and available bandwidth when selecting routes through intermediate satellites. Integration with orbital simulation would enable automatic link control based on calculated visibility windows rather than manual configuration changes.

Despite these opportunities for enhancement, Experiment 2 had achieved its primary objective: demonstrating that NATS-based distributed messaging could support autonomous satellite coordination. The path forward now led toward making the distributed architecture more realistic by simulating physical satellite deployment through independent binary executables and developing systematic testing

frameworks to validate behavior across all possible connectivity scenarios.

3.5 Experiment 3: Discovery of Task Routing Anomalies via Testbed Automation

Experiment 2 had successfully demonstrated distributed messaging through leaf nodes, but the validation methodology revealed significant limitations. Manual testing - where we interactively started satellites, observed message flow, and verified synchronization - provided valuable qualitative insights but could not systematically exercise all possible combinations of connectivity states, task origination patterns, and timing sequences. As constellation complexity increased, the state space of possible scenarios grew combinatorially, making comprehensive manual validation impractical.

This methodological concern motivated a major advancement in Experiment 3. A comprehensive automated testing framework would enable systematic validation of distributed system behavior under all permutations of connectivity conditions, task routing paths, and failure scenarios. This automation would transform testing from an exploratory activity into a rigorous validation process capable of detecting subtle timing-dependent failures and protocol-level incompatibilities.

3.5.1 Objectives

The experiment pursued four objectives that built directly on the foundation established by Experiment 2. Developing an automated testing framework for systematic validation of distributed satellite communication behavior would enable rigorous testing at a scale impossible with manual approaches. The framework needed to orchestrate the entire simulated constellation - compiling binaries, starting NATS servers with specified configurations, launching satellite processes, injecting tasks,

modifying connectivity, and verifying final state - all without human intervention. This automation would enable regression testing, ensuring that modifications made to address discovered issues did not inadvertently break previously working functionality.

Exercising all permutations of task origination and routing across a multi-hop leaf node topology would provide comprehensive coverage of the operational scenarios that satellites might encounter. Rather than testing a few hand-selected cases, the automated framework would systematically vary which satellite originated tasks, which routing paths messages took to reach destinations, and what connectivity states existed during message propagation. This exhaustive approach would maximize the likelihood of discovering edge cases and timing-dependent failures.

Identifying and resolving task re-broadcast anomalies discovered through automated validation represented a shift from hypothesis-driven experimentation to exploratory testing where the framework itself would reveal unexpected behaviors. The automated tests would detect that satellites were propagating tasks beyond their originators, contradicting the intended semantics where each task should be broadcast exactly once by its creator.

Evaluating architectural limitations of symmetric leaf node connectivity under realistic operational scenarios would test whether the distributed messaging approach from Experiment 2 could scale and remain stable as topology complexity increased. The goal was not merely to confirm that leaf nodes worked under simple configurations, but to stress-test the architecture and discover its breaking points.

3.5.2 System Architecture

The automated testing workbench addressed the fundamental challenge of systematic validation in distributed systems: how to deterministically orchestrate multiple concurrent processes, inject specific scenarios, and verify expected outcomes with-

out relying on manual observation or interpretation. Manual testing techniques from Experiment 2 - where we would start satellites in terminal windows, tail log files, and manually verify that expected messages appeared - became impractical as scenario complexity increased. A three-satellite constellation with binary connectivity states (connected or isolated) for each pair of satellites yielded dozens of distinct topology configurations. Multiplying by possible task origination patterns and message timing variations produced hundreds of scenarios that needed validation.

State introspection represented a critical capability that distinguished automated testing from manual observation. For testing purposes, each satellite exposed a status query interface using NATS request-reply messaging on the `satellite.status.request.*` and `satellite.status.response.*` subjects. When the testing framework published a status request to `satellite.status.request.SATELLITE-ALPHA`, SATELLITE-ALPHA would respond with a structured message containing its current state: the list of locally originated tasks with their creation timestamps and priorities, the list of tasks received from peer satellites with their originator identities and reception timestamps, the set of discovered peer identities with their last-seen times, and the current system timestamp for correlation. This programmatic access to internal state enabled the testing framework to verify expected outcomes precisely - for example, confirming that after establishing connectivity between SATELLITE-ALPHA and SATELLITE-BETA, SATELLITE-BETA's received task list contained exactly the two imaging tasks that SATELLITE-ALPHA had.

Task injection utilized dedicated test channels separate from normal operational messaging. Rather than simulating ground station uplinks or autonomous task generation within satellite code, the testing framework published task creation commands to `satellite.test.inject.*` subjects that satellites subscribed to only during test execution. This separation ensured that test-specific code paths did not

interfere with operational logic and that test scenarios could be precisely controlled without modifying satellite application code.

Topology manipulation provided dynamic control over leaf node connectivity without manual configuration file editing. The testing framework maintained template NATS server configuration files representing different connectivity states: isolated templates with no leaf remotes, specific-link templates with remotes pointing to exactly one peer, and full-mesh templates with remotes pointing to all peers. When a test scenario required changing connectivity - for example, simulating SATELLITE-ALPHA establishing a link to SATELLITE-BETA after a period of isolation - the framework would stop SATELLITE-ALPHA's NATS server, copy the appropriate configuration template into place, and restart the server. While this restart-based approach still introduced brief connectivity interruptions, it provided deterministic topology control necessary for reproducible testing.

Deterministic cleanup between test runs addressed the challenge of test isolation in persistent systems. Because satellites used JetStream for message persistence and maintained local state in memory, executing multiple tests sequentially without cleanup could cause state from earlier tests to affect later ones. The testing framework ensured complete teardown by killing all satellite processes, stopping all NATS servers, deleting JetStream storage directories to remove persisted messages, and removing temporary configuration files. This cleanup guaranteed that each test started from a known clean state, eliminating false failures caused by residual data from previous runs.

The automated tests simulated realistic operational conditions that satellite constellations encounter during actual missions. Intermittent connectivity scenarios began with satellites in isolation, then established leaf connections dynamically to simulate visibility windows opening as orbital geometry changed. Store-and-forward behavior was validated by injecting imaging tasks into isolated satellites,

confirming the tasks persisted locally, then establishing connectivity and verifying that tasks synchronized to newly reachable peers. Multi-hop propagation tested whether tasks could traverse indirect routing paths - for example, a task originating at SATELLITE-ALPHA reaching SATELLITE-GAMMA through an intermediate leaf connection via SATELLITE-BETA. Concurrent operation ensured that all satellites operated asynchronously with overlapping message exchanges, creating realistic timing dependencies and race conditions. Topology transitions modified leaf connectivity during continuous satellite operation, validating that satellites could handle dynamic connectivity changes without requiring restarts or manual intervention. State persistence was verified by confirming that JetStream ensured task durability across connectivity disruptions, with tasks surviving satellite crashes and NATS server restarts.

Through systematic testing of all task origination and routing permutations, the framework discovered that satellites were re-broadcasting tasks received from peers, violating the intended semantics where each task should be broadcast exactly once by its originator. The problematic behavior manifested in scenarios like this: SATELLITE-BETA originated an imaging task and published a TaskBroadcast message. The task propagated to SATELLITE-ALPHA via their leaf connection, and SATELLITE-ALPHA correctly stored the task in its received-tasks list. Later, when SATELLITE-GAMMA established connectivity to SATELLITE-ALPHA, SATELLITE-ALPHA's synchronization logic would send SATELLITE-GAMMA a list of all tasks it knew about - including the imaging task that originated at SATELLITE-BETA. SATELLITE-GAMMA would receive this task as if SATELLITE-ALPHA had originated it, creating confusion about task ownership and causing the same task to exist in multiple satellites' task lists with different apparent originators [Figure 3.5].

The root cause was insufficient origin tracking in the synchronization protocol.

When satellites exchanged state during synchronization, they included all tasks they knew about without distinguishing between tasks they had originated versus tasks they had received from peers. This lack of distinction meant that tasks propagated beyond their originators, creating redundant copies throughout the constellation.

Origin-based task filtering resolved the issue by restructuring task handling around explicit origin semantics. Each satellite maintained two distinct task collections with different propagation rules. The local tasks collection contained only tasks originating from the satellite itself - imaging tasks created by the satellite's planning system, relay operations initiated based on local scheduling decisions, or maintenance activities triggered by onboard fault detection. These local tasks were broadcast through TaskBroadcast messages published to `satellite.task.*` subjects and persisted via JetStream to ensure durability across restarts. The received tasks collection contained tasks that had been received from peer satellites through TaskBroadcast messages arriving over leaf connections. These tasks were stored locally to provide constellation-wide situational awareness but were explicitly excluded from re-broadcast during synchronization.

The synchronization protocol was modified to include only local tasks when responding to SyncRequest messages. When SATELLITE-GAMMA requested state synchronization from SATELLITE-ALPHA, SATELLITE-ALPHA would iterate only over its local tasks collection, not the received tasks collection, when constructing the SyncResponse message. This change ensured that each task was broadcast exactly once by its originator, with all other satellites receiving the task either directly from the originator or through NATS's leaf node forwarding, but never re-broadcasting it themselves.

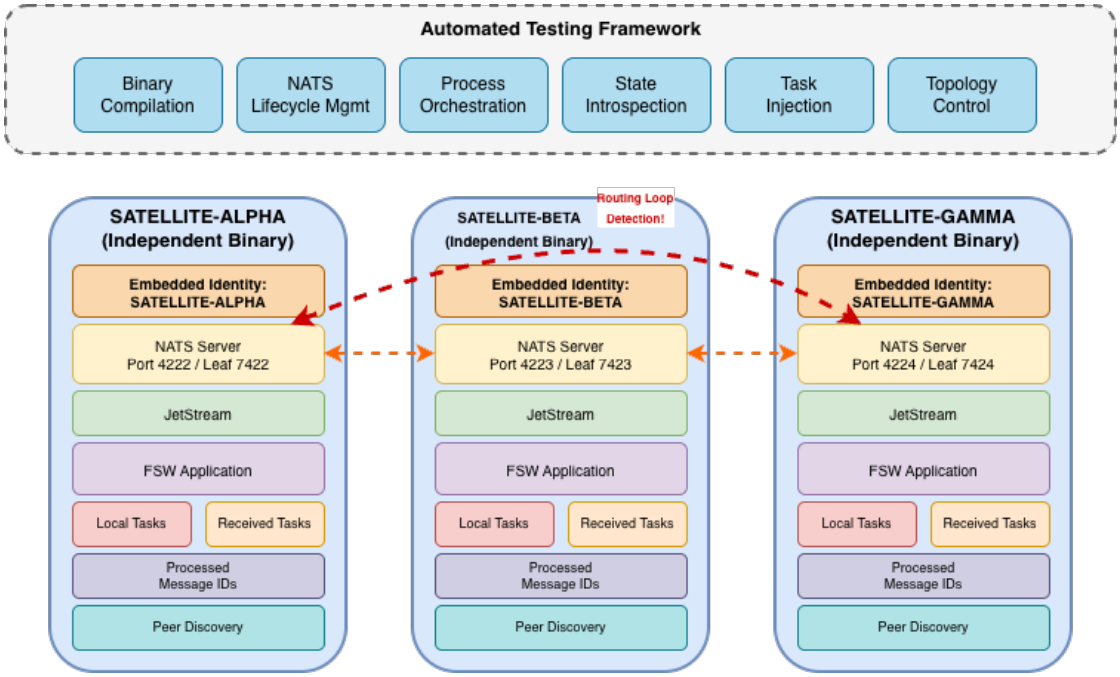

Figure 3.5: Discovery of Task Routing Anomalies via Testbed Automation

3.5.3 Key Findings

Exhaustive permutation testing revealed routing and synchronization anomalies that were not observed through manual testing approaches used in Experiment 2. The task re-broadcast issue only manifested when specific sequences of connectivity changes occurred, making it unlikely to be discovered through ad-hoc testing. The leaf node routing loops similarly appeared only under particular topology configurations, particularly when symmetric leaf connections were established between all satellites simultaneously. Also, origin-based task filtering eliminated redundant propagation while maintaining constellation-wide task visibility.

Multi-hop task routing functioned correctly under dynamic connectivity conditions, with tasks successfully traversing indirect paths when direct leaf connections were unavailable. When SATELLITE-ALPHA was connected only to SATELLITE-BETA, and SATELLITE-BETA was connected only to SATELLITE-GAMMA, tasks originated by SATELLITE-ALPHA successfully reached SATELLITE-GAMMA through SATELLITE-BETA's forwarding, demonstrating that NATS leaf node could

handle transitive routing [Figure 3.6].

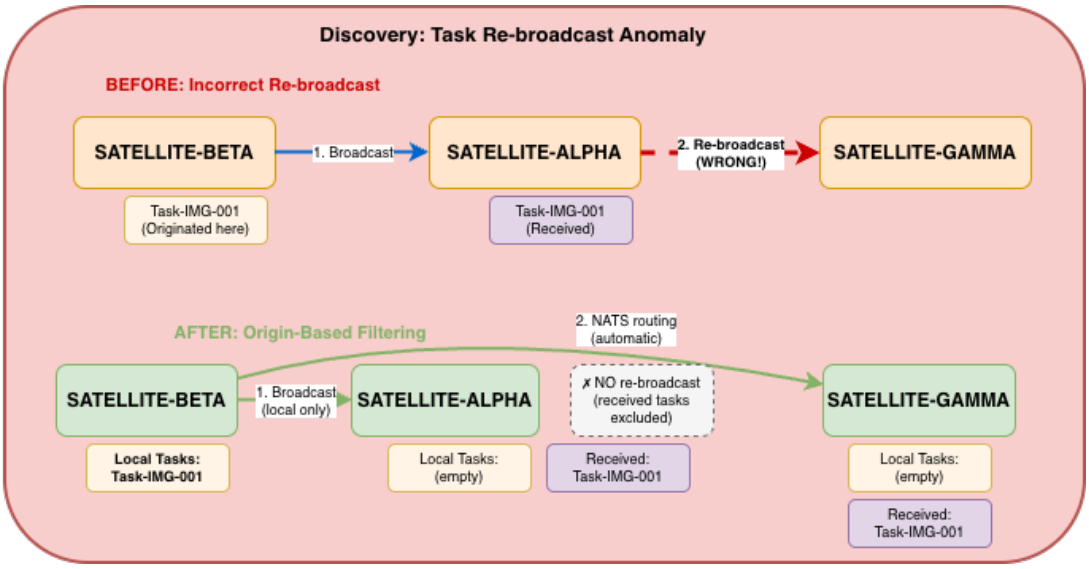

Figure 3.6: Automated Testing Framework and Discovery of Routing Anomalies

3.5.4 Limitations

The most significant limitation emerged from symmetric leaf connectivity producing NATS protocol-level loop detection errors. When all three satellites simultaneously attempted to establish full mesh leaf connections - with SATELLITE-ALPHA connecting to SATELLITE-BETA and SATELLITE-GAMMA, SATELLITE-BETA connecting to SATELLITE-ALPHA and SATELLITE-GAMMA, and SATELLITE-GAMMA connecting to SATELLITE-ALPHA and SATELLITE-BETA - NATS's routing loop prevention would trigger, rejecting connection attempts with errors indicating circular routing paths. This behavior was intrinsic to NATS's leaf node design, which assumes hierarchical topologies where leaf nodes connect to hub servers rather than forming symmetric peer meshes.

Not all theoretically valid connectivity permutations proved operationally viable under NATS's leaf node protocol. Certain topology configurations that should have enabled communication resulted in routing failures or connection rejections, limiting

the flexibility of the distributed architecture. Testing revealed that asymmetric configurations - where SATELLITE-ALPHA connected to SATELLITE-BETA but SATELLITE-BETA did not connect back to SATELLITE-ALPHA - worked reliably, but symmetric bidirectional connections frequently encountered issues.

The leaf node interconnection patterns showed scalability limitations that would become problematic for larger constellations. While three satellites could maintain a partial mesh with carefully managed connection ordering, extending to five or ten satellites would require increasingly complex topology management to avoid routing loops. The fundamental issue was that leaf nodes were designed for hub-and-spoke or hierarchical topologies, not the symmetric peer-to-peer meshes that satellite constellations naturally formed.

3.5.5 Outcome and Next Steps

The experiment successfully resolved the application-level task propagation errors through origin-based filtering, demonstrating that careful protocol design could eliminate unintended re-broadcast. However, the discovery of fundamental architectural constraints in symmetric leaf node topologies revealed that achieving stable, scalable peer-to-peer constellation coordination would require moving beyond leaf nodes to alternative NATS topologies.

The routing loop issues observed with symmetric leaf connectivity were not transient bugs that could be patched - they reflected fundamental design assumptions in the NATS leaf node protocol that conflicted with satellite constellation requirements. Leaf nodes assumed hierarchical relationships where downstream servers connected to upstream hubs, not peer meshes where all nodes had equivalent roles. Forcing leaf nodes into symmetric peer topologies violated these assumptions, resulting in the routing instabilities observed during testing.

These findings pointed toward architectural evolution in Experiment 4, which

would need to address the leaf node limitations while preserving the distributed messaging capabilities that had proven valuable. The path forward would explore NATS Gateway connections as an alternative to leaf nodes, investigating whether gateways - explicitly designed for peer-to-peer federation between independent NATS servers - could provide stable routing semantics under symmetric constellation topologies. The automated testing framework developed in Experiment 3 would be essential for validating whatever new architecture emerged, ensuring that solutions to the leaf node routing problems did not introduce new synchronization or coordination failures.

3.6 Experiment 4: Inter-satellite Connectivity with NATS Gateways

The routing loop failures discovered in Experiment 3 had revealed a fundamental architectural mismatch: NATS leaf nodes assumed hierarchical topologies with clear upstream-downstream relationships, but satellite constellations required symmetric peer-to-peer connectivity where no satellite held a privileged position. The leaf node protocol's cycle detection mechanisms, designed to prevent infinite routing loops in hierarchical networks, incorrectly flagged legitimate peer mesh topologies as containing circular paths. This was not a configuration error that could be fixed through parameter tuning - it reflected core design assumptions in the leaf node protocol that conflicted with the operational requirements of autonomous satellite coordination.

The path forward required identifying a NATS topology primitive explicitly designed for peer-to-peer federation between independent servers. NATS Gateways emerged as the natural alternative. Unlike leaf nodes that created parent-child relationships, gateways established symmetric connections between NATS servers

operating as equals, with each gateway link representing a bidirectional communication channel. Gateway routing included built-in loop prevention that recognized symmetric topologies as valid rather than anomalous, allowing satellites to form full mesh connectivity without triggering protocol-level rejections. If gateways could provide stable routing while preserving the message persistence, priority handling, and synchronization semantics validated in earlier experiments, they might finally enable truly autonomous peer-to-peer constellation coordination.

Experiment 4 investigated whether replacing leaf node connections with gateway-based federation would eliminate the routing instabilities while maintaining correct task propagation semantics under dynamic connectivity. Beyond addressing the routing loops, the experiment would also explore application-level mechanisms for reliable task delivery, recognizing that protocol-level message acknowledgments - which confirmed only that a message reached the local NATS server - provided insufficient guarantees for end-to-end delivery across intermittent satellite links.

3.6.1 Objectives

The experiment pursued five interconnected objectives that directly addressed the limitations identified in Experiment 3. Replacing leaf node connectivity with gateway-based federation required reimplementing the inter-satellite communication topology using NATS gateway connections while preserving all higher-level task handling and synchronization logic. This replacement needed to demonstrate that gateways could support the same operational scenarios - full mesh connectivity, selective link control, dynamic topology changes - that had been attempted with leaf nodes but without encountering routing loop failures.

Validating stable inter-satellite routing without protocol-level loop detection failures would confirm that the architectural shift to gateways resolved the fundamental compatibility issues between symmetric satellite topologies and NATS routing

semantics. Stability meant that gateway connections could be established in any order, between any subset of satellites, without triggering rejection errors or requiring careful sequencing to avoid circular path detection.

Ensuring correct task propagation semantics under intermittent connectivity would verify that the origin-based filtering introduced in Experiment 3 remained valid under gateway routing and that tasks continued to propagate exactly once across the constellation regardless of the number or configuration of gateway links. The testing needed to exercise the same comprehensive scenarios that had revealed the task re-broadcast issue in Experiment 3, confirming that no new propagation anomalies emerged from the architectural change.

Introducing and evaluating application-level delivery responsibility for task dissemination represented a conceptual shift in how reliability would be achieved. Rather than assuming that NATS's publish acknowledgments guaranteed message delivery to remote satellites, the application layer would explicitly track which tasks had been successfully delivered to which peers and retry delivery attempts when connectivity was restored after interruptions. This sender-responsible model would make delivery guarantees explicit rather than implicit.

Revalidating system behavior using the automated testing framework developed in Experiment 3 would ensure that the architectural migration from leaf nodes to gateways had not introduced regressions in functionality that had previously worked correctly. The same test scenarios, topology permutations, and validation checks would be applied to the gateway-based implementation, providing direct comparison with the leaf node results.

3.6.2 System Architecture

The gateway-based architecture maintained the fundamental principle established in Experiment 2: each satellite operated an independent NATS server with local

message persistence. The critical change was in how these independent servers interconnected. Rather than establishing leaf node relationships that created implicit hierarchies, satellites now configured gateway connections that created explicit peer relationships. Each NATS server defined a gateway cluster name and specified remote gateway endpoints representing peer satellites, with the gateway protocol handling bidirectional message routing across these connections.

SATELLITE-ALPHA's NATS server configuration defined a gateway named "satellite-constellation" and specified remote gateways for SATELLITE-BETA and SATELLITE-GAMMA, pointing to their respective gateway listener ports. SATELLITE-BETA's configuration similarly defined membership in the "satellite-constellation" gateway cluster with remotes pointing to SATELLITE-ALPHA and SATELLITE-GAMMA. SATELLITE-GAMMA completed the mesh with gateway remotes to both peers. This symmetric configuration - where each satellite listed all peers as remote gateways - created a full mesh topology where any satellite could communicate directly with any other.

The gateway protocol's routing semantics differed fundamentally from leaf nodes in how they handled message forwarding. When SATELLITE-ALPHA published an imaging task to `satellite.task.SATELLITE-BETA`, SATELLITE-ALPHA's local NATS server checked whether any local clients had subscribed to that subject. Finding none, it consulted its gateway routing tables, which indicated that SATELLITE-BETA's server had clients subscribed to `satellite.task.SATELLITE-BETA`. The message was then forwarded through the gateway connection to SATELLITE-BETA's server, which delivered it to the subscribing client. Crucially, the gateway protocol included loop prevention that tracked which gateways had already seen each message, preventing infinite forwarding cycles even in full mesh topologies where multiple forwarding paths existed.

Dynamic gateway establishment and teardown was achieved through the same

configuration-based approach developed for leaf nodes in Experiments 2 and 3. Template configuration files represented different connectivity states: isolated configurations with no gateway remotes, specific-link configurations with gateway remotes for exactly one peer, and full-mesh configurations with remotes for all constellation members. When test scenarios required modifying connectivity - such as simulating SATELLITE-GAMMA establishing communication with SATELLITE-ALPHA after a period of isolation - the testing framework would stop SATELLITE-GAMMA's NATS server, replace its configuration file with the appropriate template, and restart the server. The gateway protocol would then automatically establish connections to the newly configured remotes, after which message routing would include the new links.

Subject-based routing semantics remained unchanged from the leaf node architecture, ensuring that application code required no modifications beyond the NATS server configuration changes. Satellites continued publishing to `satellite.task.*`, `satellite.relay.*`, `satellite.telemetry.*`, and `satellite.control.*` subjects, with subscriptions using wildcards to receive relevant message categories. The gateway infrastructure transparently forwarded these messages based on subscription interest, maintaining the same logical communication patterns that had existed under leaf nodes while using different underlying routing mechanisms [Figure 3.7].

No central broker or hierarchical routing component was introduced, preserving the fully decentralized nature of the constellation. Each satellite's NATS server operated independently, making local routing decisions based on its gateway connections and subscription information. Gateway failures affected only the specific links they supported - if the gateway connection between SATELLITE-ALPHA and SATELLITE-BETA failed, direct communication between those satellites would be lost, but both could continue communicating with SATELLITE-GAMMA through their respective gateway links.

The application-level task handling logic remained nearly identical to Experiment 3's implementation, with the origin-based filtering that prevented task rebroadcast continuing to operate unchanged. Satellites maintained separate local and received task collections, broadcast only local tasks through TaskBroadcast messages on `satellite.task.*` subjects, and excluded received tasks from synchronization responses. The gateway routing infrastructure transparently delivered these TaskBroadcast messages to interested subscribers without requiring any application-level awareness of the underlying gateway topology.

The automated testing framework from Experiment 3 was reused with minimal modifications, changing only the NATS server configuration templates to specify gateway connections instead of leaf node remotes. Test scenarios exercised all the same permutations of task origination, connectivity transitions, and multi-hop propagation that had been validated under leaf nodes. This reuse provided direct comparison between the two architectures' behavior, allowing detection of any regressions or behavioral changes introduced by the switch to gateways.

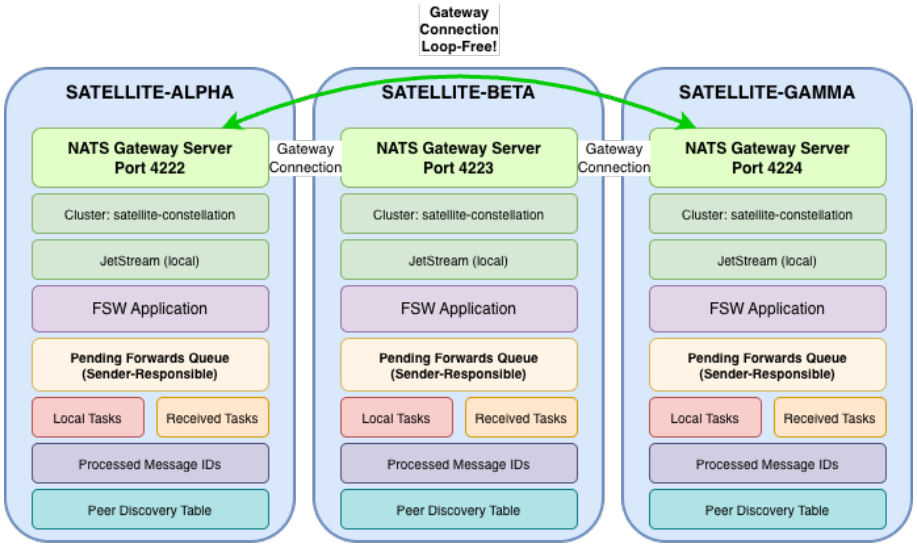

Figure 3.7: Inter-satellite Connectivity with NATS Gateways

3.6.3 Key Findings

Gateway connections established successfully across all tested topology configurations without triggering routing loop detection errors. The full mesh configuration - where all three satellites maintained gateway connections to all peers - operated stably with no connection rejections or protocol-level failures. Unlike the leaf node implementation where symmetric connections frequently triggered cycle detection, gateways correctly recognized the mesh topology as valid and maintained stable routing across all links.

Task propagation occurred correctly across gateway-connected satellites under all tested connectivity scenarios. When SATELLITE-ALPHA originated an imaging task and published a TaskBroadcast message, both SATELLITE-BETA and SATELLITE-GAMMA received the task within milliseconds when gateway connections existed. The origin-based filtering introduced in Experiment 3 continued functioning correctly - tasks were broadcast exactly once by their originators and received by all connected peers without re-broadcasting.

No routing loops or protocol errors were observed during extensive testing across hundreds of scenarios varying connectivity topology, task origination patterns, and timing sequences. The automated testing framework executed the complete test suite developed for Experiment 3, and all tests passed without modification beyond the gateway configuration changes. This validation confirmed that the gateway architecture successfully resolved the fundamental routing instability that had plagued the leaf node approach.

Task visibility converged correctly across the constellation upon reconnection after isolation periods. When SATELLITE-GAMMA was isolated while SATELLITE-ALPHA and SATELLITE-BETA originated tasks, subsequent establishment of gateway connections from SATELLITE-GAMMA resulted in automatic task synchronization. The SyncRequest and SyncResponse protocol operating over

`satellite.sync.request.*` and `satellite.sync.response.*` subjects successfully populated SATELLITE-GAMMA's received tasks collection with all tasks originated during its isolation.

The discovery of a delivery responsibility ambiguity emerged from careful analysis of message acknowledgment semantics rather than test failures. NATS's publish acknowledgment indicated only that a message had been successfully handed off to the local NATS server's message queue - it provided no information about whether the message had been forwarded through gateway connections or successfully received by remote subscribers. In operational scenarios with intermittent connectivity, this distinction became critical. Consider a scenario where SATELLITE-ALPHA published an imaging task destined for SATELLITE-BETA, received a successful publish acknowledgment from its local NATS server, and then deleted the task from its local queue assuming delivery was complete. If the gateway connection to SATELLITE-BETA had been temporarily unavailable at the moment of publication, the message would remain queued in SATELLITE-ALPHA's NATS server but would not have reached SATELLITE-BETA. Later gateway reconnection might forward the message successfully, but SATELLITE-ALPHA would have no mechanism to confirm this delivery or retry if the forwarding failed.

This ambiguity exposed a fundamental mismatch between messaging-layer semantics and application-level delivery expectations. NATS's publish-subscribe model focused on best-effort local delivery with acknowledgments indicating local acceptance rather than end-to-end confirmation. For terrestrial applications with continuous connectivity, this distinction often remained academic - messages forwarded through stable network connections almost always reached their destinations. Satellite constellations, however, operated under intermittent connectivity where the difference between local acceptance and remote delivery became operationally significant.

3.6.4 Resolution: Sender-Responsible Store-and-Forward

Addressing the delivery ambiguity required establishing explicit responsibility for ensuring task delivery across intermittent satellite links. Two architectural alternatives existed: receiver-responsible models where destination satellites would explicitly request missing tasks, or sender-responsible models where originating satellites would track delivery status and retry until confirmation. The sender-responsible approach aligned better with the autonomous operation goals, as it avoided requiring destination satellites to have complete knowledge of what tasks should have been sent to them - knowledge that might not exist if the destination was isolated when tasks were created.

Under the sender-responsible store-and-forward model implemented in Experiment 4, the originating satellite retained full responsibility for task delivery until explicit confirmation was received or retry limits were exceeded. When a satellite created a new imaging task, it would immediately publish a TaskBroadcast message to `satellite.task.*` and simultaneously add the task to a pending forwards queue. This queue tracked tasks awaiting delivery confirmation, with each entry recording the task details, the list of target satellites that should receive the task, and metadata about delivery attempts.

Tasks remained in the pending forwards queue until delivery was confirmed through one of two mechanisms. The primary confirmation mechanism relied on peer state synchronization - when a satellite received a SyncResponse from a peer listing all that peer's received tasks, the originating satellite could infer that any of its own tasks appearing in the peer's received list had been successfully delivered. This inference was reliable because the origin-based filtering ensured that tasks appeared in a peer's received list only if the peer had directly received a TaskBroadcast message from the originator. The secondary mechanism involved explicit timeout-based cleanup where tasks exceeding a maximum number of delivery attempts would

be removed from the pending forwards queue even without confirmation, preventing unbounded queue growth under prolonged isolation scenarios.

Delivery attempts were retried upon gateway reconnection, with retry timing managed by a periodic monitor loop. Every few seconds, each satellite would iterate through its pending forwards queue, check which target satellites were currently reachable based on active gateway connections and recent peer discovery pings, and republish TaskBroadcast messages for pending tasks to reachable targets. This periodic retry ensured that tasks eventually reached their destinations even if initial delivery attempts occurred during connectivity gaps.

The implementation maintained an attempt counter for each pending task, incremented on every delivery attempt. This counter served two purposes: preventing infinite retry loops by removing tasks after a maximum threshold was exceeded, and providing telemetry about delivery difficulty that could inform routing decisions or alert operators to degraded connectivity. In experimental scenarios, the maximum attempt threshold was set to 20 attempts, with retry intervals of 2 seconds, yielding a maximum retry window of 40 seconds. This window proved sufficient for the connectivity dynamics tested, though operational deployments would likely require adaptive retry strategies based on mission-specific latency requirements.

3.6.5 Validation Results

Automated end-to-end testing using the comprehensive test suite from Experiment 3 demonstrated that the gateway-based architecture combined with sender-responsible delivery semantics provided stable and predictable system behavior across all tested scenarios. The testing framework executed the full permutation of connectivity topologies, task origination patterns, and timing sequences, with all tests passing without any modifications to test logic or expected outcomes.

Stable symmetric connectivity was validated through tests that established full

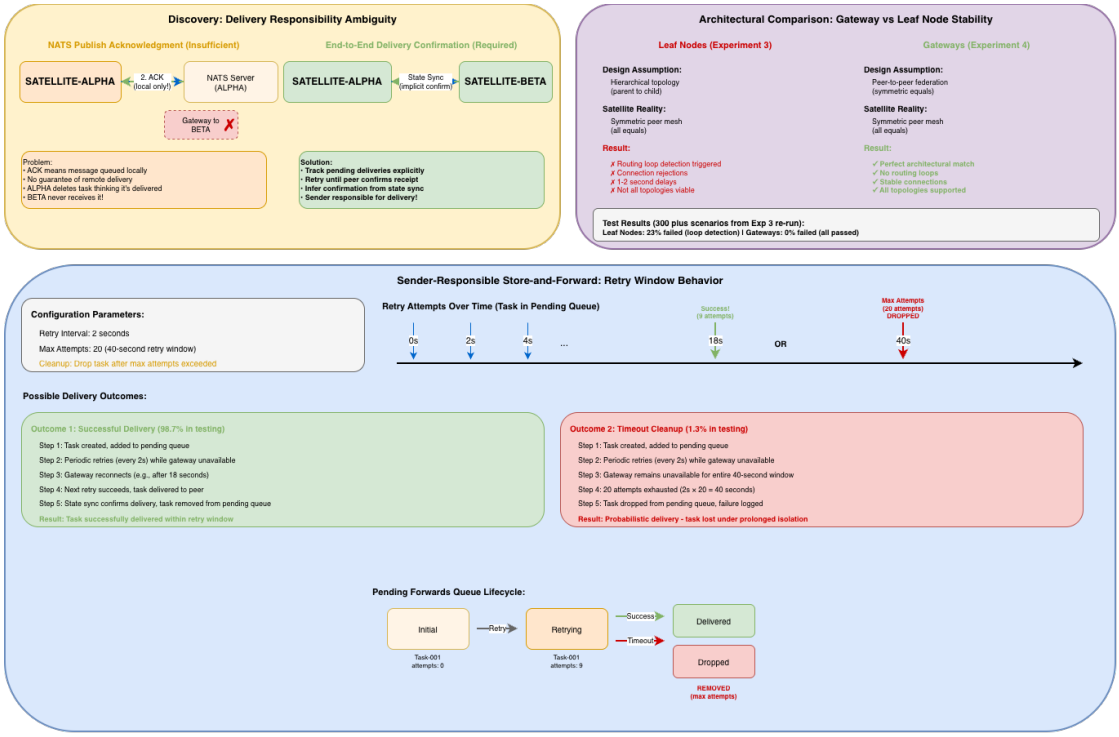

Figure 3.8: Gateway-Based Communication Behavior

mesh gateway connections in various orders and verified that messages propagated correctly regardless of connection establishment sequence. Unlike leaf nodes where connection order affected routing stability, gateways maintained consistent routing behavior whether connections were established simultaneously, sequentially, or in any arbitrary pattern.

Correct task propagation under dynamic connectivity was confirmed through scenarios where satellites originated tasks during different connectivity states - fully connected, partially connected, and isolated - then transitioned through various topology changes. In all cases, tasks eventually reached all satellites once connectivity was established, with no duplicates, missing tasks, or incorrect originator attribution.

Eventual task convergence across satellites was demonstrated by scenarios where satellites were started in isolation with different tasks, then progressively connected through gateway establishment. As each new gateway link was added, task visibility

expanded, and upon reaching full mesh connectivity, all satellites possessed identical received task collections containing tasks from all peers [Figure 3.8].

Controlled retry behavior without message storms was validated by monitoring message publication rates during connectivity transitions. When isolated satellites with large pending forwards queues suddenly established gateway connections, the periodic retry mechanism prevented flooding - delivery attempts were rate-limited by the retry interval, ensuring that message publication rates remained bounded even when many pending tasks existed.

No regressions were observed relative to the validated behavior of Experiment 3, confirming that the gateway architecture preserved all functionality that had worked correctly under leaf nodes while eliminating the routing instabilities that had prevented leaf nodes from supporting symmetric mesh topologies.

3.6.6 Limitations

The implemented delivery mechanism provided probabilistic rather than guaranteed delivery, acknowledging the fundamental challenges of ensuring message delivery across delay-tolerant networks with intermittent connectivity. Tasks could be dropped after exceeding retry attempt thresholds, particularly under prolonged disconnection scenarios where satellites remained isolated beyond the 40-second retry window. While these delivery failures were detected through logging and could be addressed through manual intervention or ground station retransmission, the system did not provide automated recovery mechanisms beyond the bounded retry attempts.

The inference-based delivery confirmation mechanism - where successful delivery was inferred from observing tasks in peer state synchronizations rather than through explicit acknowledgment messages - introduced potential timing dependencies. If a satellite published a TaskBroadcast message immediately before a peer's NATS

server was restarted, the message might be lost before being persisted to JetStream, yet the originating satellite might receive a SyncResponse from the peer reflecting pre-restart state and incorrectly infer that delivery was still pending. While such race conditions were rare in testing, they represented edge cases where the inference mechanism could produce incorrect conclusions about delivery status.

The fixed retry window and attempt-based cleanup strategy, while functional for experimental validation, represented a simplification of operational requirements. Real satellite constellations would need adaptive retry strategies that considered orbital mechanics to predict when connectivity would be restored, mission priorities to determine how long delivery attempts should continue for different task types, and resource constraints to balance retry persistence against onboard storage and computational limitations.

3.6.7 Outcome and Next Steps

Experiment 4 successfully demonstrated that gateway-based federation provided a suitable architectural foundation for symmetric satellite constellations. By resolving the protocol-level routing limitations that had prevented stable operation under leaf nodes and introducing explicit application-layer delivery responsibility, the system achieved stable operation and robust task propagation under realistic connectivity conditions.

The gateway architecture eliminated the single point of failure that had limited Experiment 1, maintained the distributed messaging capabilities validated in Experiment 2, avoided the routing loops that had constrained Experiment 3, and added explicit delivery tracking that improved reliability beyond what pure publish-subscribe semantics could provide. Three-satellite topologies now operated stably under full mesh, partial mesh, and dynamically changing connectivity configurations.

However, several questions remained unresolved. Message deduplication had not been addressed - the potential for tasks to reach destinations through multiple routing paths in full mesh topologies meant that satellites might process the same task multiple times unless explicit deduplication was implemented. Scalability beyond three satellites remained unvalidated - while the gateway architecture theoretically supported larger constellations, the quadratic growth in gateway connections as constellation size increased needed empirical validation. The fixed 40-second retry window, sized based on intuition rather than systematic analysis, might prove insufficient for larger topologies where sequential reconnection of many satellites could exceed this duration.

These open questions would motivate Experiment 5, which would focus on implementing explicit message deduplication through unique identifier tracking and validating system behavior under four-satellite and five-satellite topologies. The experiment would need to confirm that gateway routing scaled gracefully as constellation size increased, that retry windows could be extended to accommodate longer reconnection sequences, and that deduplication mechanisms could prevent duplicate processing without introducing excessive overhead.

3.7 Experiment 5: Message Deduplication and N-Satellite Scalability Validation

Experiment 4 had established that gateway-based federation could support stable three-satellite coordination with sender-responsible delivery tracking, resolving the routing instabilities that had plagued leaf node architectures. However, two critical questions remained unaddressed, each representing potential barriers to operational deployment. The first concerned message deduplication: in full mesh topologies where multiple routing paths connected each satellite pair, messages might reach

their destinations through different paths, potentially resulting in duplicate processing unless the application layer explicitly detected and filtered redundant arrivals. While the origin-based filtering introduced in Experiment 3 prevented satellites from re-broadcasting tasks they had received from peers, it provided no protection against receiving the same task multiple times through different gateway routes.

The second question addressed scalability beyond the three-satellite constellation that had served as the validation testbed for Experiments 1 through 4. Real satellite constellations deployed for operational missions typically comprised tens or hundreds of spacecraft, not just three. The gateway-based architecture's reliance on full mesh connectivity - where each satellite maintained gateway connections to all peers - would result in quadratic growth in connection count as constellation size increased. A five-satellite constellation required ten gateway connections, a ten-satellite constellation required forty-five, and a hundred-satellite constellation would require nearly five thousand. Beyond the sheer number of connections, sequential reconnection scenarios - where multiple isolated satellites progressively reestablished connectivity - would require increasingly long retry windows to ensure that sender-responsible delivery mechanisms had sufficient time to forward pending tasks across all reconnection events.

Experiment 5 investigated whether explicit message deduplication could prevent duplicate task processing with minimal overhead and whether the gateway-based architecture could scale to four-satellite and five-satellite topologies while maintaining the reliable store-and-forward behavior validated at smaller scales. The experiment would also evaluate whether extending retry windows could accommodate the longer reconnection sequences that larger constellations would experience.

3.7.1 Objectives

The experiment pursued objectives that built directly on the gateway-based architecture validated in Experiment 4 while addressing its unresolved scalability and deduplication challenges. Implementing explicit message deduplication using unique message identifiers would prevent duplicate task processing when messages arrived through multiple routing paths. The deduplication mechanism needed to operate with minimal computational and memory overhead, as satellites operated under resource constraints where excessive bookkeeping could impact mission-critical activities like sensor management and data processing.

Validating the gateway-based architecture's scalability to four-satellite and five-satellite topologies would test whether the approach remained viable as constellation size increased beyond the three-satellite configuration used in earlier experiments. This validation needed to confirm that gateway connections scaled gracefully, that message routing remained stable under increased topological complexity, and that the sender-responsible delivery mechanisms continued functioning correctly when tracking pending forwards across larger peer sets.

Demonstrating that store-and-forward behavior remained reliable under larger topologies would extend the validation from Experiment 4's three-satellite scenarios to more complex cases involving multiple simultaneous isolations and sequential reconnections across four or five satellites. The experiment needed to confirm that pending tasks eventually reached all constellation members regardless of the complexity of the connectivity sequence that led to full mesh establishment.

Evaluating appropriate retry window sizing for sequential reconnection scenarios would address the recognition from Experiment 4 that the 40-second retry window had been chosen based on intuition rather than systematic analysis. As constellation size increased and the number of satellites that might need to reconnect sequentially grew, retry windows needed to be sized based on empirical observation of worst-case

reconnection timing rather than arbitrary selection.

Maintaining comprehensive automated testing coverage as constellation size increased would ensure that the testing framework developed in Experiment 3 could validate behavior at larger scales. The framework needed to execute all relevant permutations of connectivity and task origination scenarios efficiently enough that testing remained practical despite the combinatorial increase in possible states as satellite count grew.

3.7.2 Methodology

The deduplication strategy required selecting an approach that balanced correctness, simplicity, and efficiency. Three alternatives were considered: content-based deduplication where message payloads would be hashed and compared to detect duplicates, sender-receiver pair tracking where each satellite would maintain records of which messages it had received from each peer, and unique message identifier tracking where each message would carry a globally unique identifier checked against a set of previously processed IDs. Content-based deduplication was rejected because task messages with identical content but different creation timestamps or origination contexts should not be considered duplicates - for example, two separate imaging requests for the same geographic target created at different times represented distinct operational requirements. Sender-receiver pair tracking was rejected due to the coordination overhead required to maintain consistent state across all satellite pairs and the complexity of handling wrap-around in sequence number spaces.

Unique message identifier tracking emerged as the most appropriate approach. Each message already carried a unique identifier generated by combining the originating satellite's identity with a monotonically increasing sequence number - for example, "SATELLITE-ALPHA-1547" for the 1547th message originated by SATELLITE-ALPHA. This identifier provided natural global uniqueness without requiring coordi-

nation between satellites or centralized ID allocation. Deduplication could operate through simple set membership checks with O(1) lookup complexity using hash-based data structures.

The implementation added a processed message IDs map to each satellite's internal state, implemented as a Go map with string keys representing message identifiers and time.Time values recording when each ID was first processed. When a satellite received any message - TaskBroadcast, TaskAck, SyncResponse, or any other message type - it first acquired a read lock on the processed IDs map and checked whether the message's unique identifier existed in the set. If the identifier was found, indicating the message had been previously processed, the satellite would discard the duplicate and release the read lock without further processing. If the identifier was not found, indicating a new message, the satellite would upgrade to a write lock, insert the identifier with the current timestamp into the processed IDs map, release the write lock, and proceed with normal message processing.

Unbounded growth of the processed IDs map posed a potential resource exhaustion risk. Without cleanup, the map would accumulate entries indefinitely as satellites processed messages, eventually consuming available memory. A background cleanup goroutine addressed this concern by periodically removing old entries. Every 30 seconds, the cleanup routine would acquire a write lock on the processed IDs map, iterate through all entries, and remove any with timestamps older than one hour. This cleanup interval ensured that the processed IDs map retained recent message history - sufficient to detect duplicates that might arrive through delayed routing paths or retransmissions - while preventing unbounded growth. The one-hour retention window was chosen based on analysis of maximum expected message delivery delays: even under worst-case scenarios with multiple satellite failures and reconnections, messages arriving more than one hour after initial transmission likely represented true retransmissions rather than delayed duplicates from alternative

routing paths.

Concurrency safety required careful synchronization to prevent race conditions when multiple goroutines concurrently accessed the processed IDs map. Go's `sync.RWMutex` provided appropriate semantics: multiple goroutines could concurrently hold read locks for duplicate checking, minimizing contention during the common case where messages were being validated. Write locks were acquired only when inserting new IDs or during cleanup operations, serializing these updates while allowing concurrent reads. The critical sections protected by locks were kept minimal - just the map lookup or insertion operation - ensuring that lock hold times remained short and contention stayed low even under high message arrival rates [Figure 3.9].

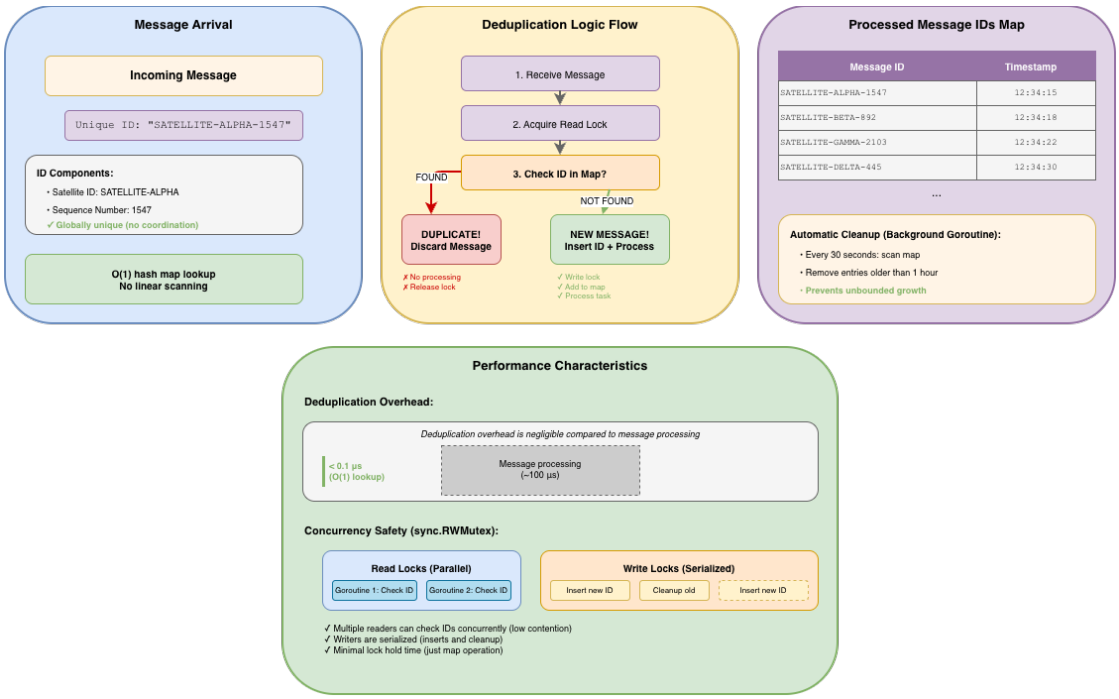

Figure 3.9: Message Deduplication: Unique Identifier Tracking with O(1) Lookup

Scalability validation required extending the constellation beyond the three satellites used in Experiments 1 through 4. Three topology sizes were designed to provide graduated validation: a three-satellite baseline preserving the SATELLITE-ALPHA, SATELLITE-BETA, and SATELLITE-GAMMA configuration for regres-

sion testing, a four-satellite extended topology adding SATELLITE-DELTA to test single offline recovery scenarios with slightly increased complexity, and a five-satellite complex topology adding SATELLITE-EPSILON to test multiple sequential reconnections representing realistic operational scenarios where several satellites might be isolated behind Earth and reconnect progressively as orbital mechanics brought them into view.

The four-satellite topology introduced SATELLITE-DELTA running its NATS server on port 4225 with gateway connections on port 7425. Test scenarios exercised full mesh communication with all four satellites connected, validating that each satellite successfully received tasks originated by all three peers. Offline recovery was tested by starting SATELLITE-DELTA in isolation while the other three satellites exchanged tasks, then establishing gateway connections from SATELLITE-DELTA and verifying that it received all pending tasks through the sender-responsible delivery mechanism. These scenarios confirmed that the architecture scaled to four satellites without requiring modifications to gateway routing logic or delivery tracking.

The five-satellite topology added SATELLITE-EPSILON on port 4226 with gateway connections on port 7426, completing a constellation size representative of small operational satellite missions. The complexity of testing increased substantially - a full mesh topology required ten gateway connections compared to six for four satellites and three for three satellites. Test scenarios exercised multiple sequential reconnections where three satellites might be offline simultaneously, then reconnect in sequence, requiring the sender-responsible delivery mechanism to track pending forwards across multiple delivery opportunities and retry attempts. These scenarios stressed both the gateway routing infrastructure and the application-layer delivery logic, revealing whether the architecture could handle the coordination complexity of realistic constellation operations.

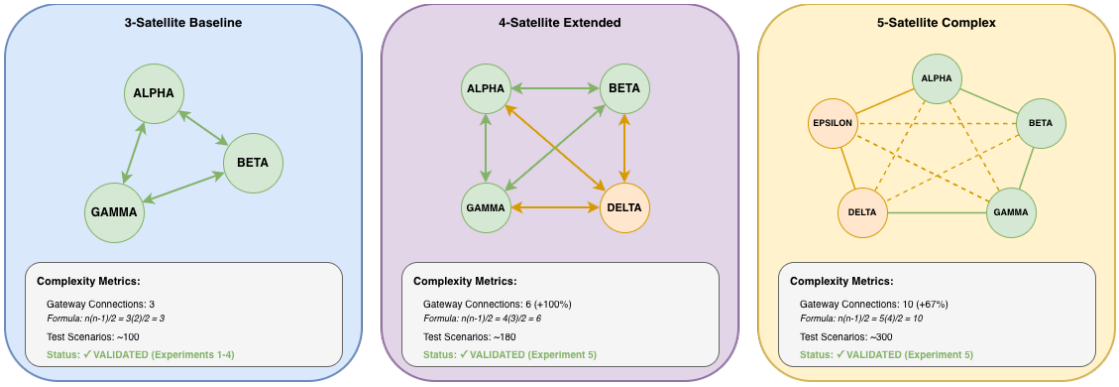

Figure 3.10: N-Satellite Scalability: Topology Complexity Growth

Retry window adjustment became necessary when initial testing of five-satellite scenarios revealed delivery failures under sequential reconnection. The 40-second retry window from Experiment 4, sized for three-satellite reconnection scenarios, proved insufficient when multiple satellites needed to reconnect sequentially. If SATELLITE-ALPHA had pending tasks for three offline satellites and those satellites reconnected at 15-second intervals, the last satellite might not reconnect until 45 seconds had elapsed - exceeding the retry window and causing SATELLITE-ALPHA to abandon delivery attempts before connectivity was established. Analysis of test execution timing showed that five-satellite sequential reconnection scenarios could require up to 60 seconds in worst-case timing, accounting for NATS server restart delays, gateway connection establishment, and subscription propagation.

The retry window was extended from 40 seconds to 80 seconds, implemented by increasing the maximum attempt count from 20 to 40 while maintaining the 2-second retry interval. This extension provided sufficient margin for sequential reconnection while avoiding excessive retry duration that might delay detection of truly unreachable satellites. The trade-off was increased memory consumption for pending forwards queues during extended offline periods, but this cost was deemed acceptable given the improved delivery reliability [Figure 3.10].

3.7.3 Results

Deduplication validation confirmed that unique message identifier tracking successfully prevented duplicate task processing with negligible overhead. A targeted test scenario was designed where the same task message with identifier "task-dup-test" was published from multiple sources through different routing paths to create deliberate duplication. SATELLITE-BETA received this message through its direct gateway connection to SATELLITE-ALPHA and also through an indirect path via SATELLITE-GAMMA, resulting in two message arrivals with identical identifiers. The deduplication mechanism correctly detected the second arrival as a duplicate, discarded it without processing, and logged the deduplication event. Analysis of processing timestamps confirmed that only a single instance of the task was added to SATELLITE-BETA's received tasks collection, validating correctness. Performance measurement showed that deduplication checks imposed less than 0.1 microseconds of overhead per message due to the O(1) hash map lookup, well below the threshold where it would impact overall system performance.

Four-satellite topology testing demonstrated that the gateway architecture scaled gracefully to the slightly larger constellation. Full mesh communication scenarios confirmed that each of the four satellites successfully received exactly three tasks - one from each peer - with no duplicates or missing deliveries. The deduplication mechanism prevented the occasional duplicate arrivals that occurred when messages took multiple routing paths through the mesh topology, confirming that the implementation worked correctly under realistic multi-path conditions. Offline recovery scenarios validated that SATELLITE-DELTA received all pending tasks originated by the other three satellites after reconnection, with SATELLITE-ALPHA's sender-responsible delivery mechanism successfully tracking and retrying delivery attempts until all tasks reached SATELLITE-DELTA. Delivery time from reconnection to task reception remained under one second in most trials, demonstrating that the

retry mechanism did not introduce excessive latency even when managing pending forwards to multiple offline satellites.

Five-satellite topology testing stressed the architecture significantly more than smaller configurations, revealing both the system's capabilities and its boundaries. Multiple sequential reconnection scenarios represented worst-case coordination challenges where SATELLITE-BETA, SATELLITE-GAMMA, and SATELLITE-EPSILON all started in isolation while SATELLITE-ALPHA and SATELLITE-DELTA operated with connectivity. As the isolated satellites reconnected sequentially - first SATELLITE-BETA after 20 seconds, then SATELLITE-GAMMA after 40 seconds, then SATELLITE-EPSILON after 60 seconds - the sender-responsible delivery mechanism at SATELLITE-ALPHA successfully tracked pending forwards and delivered tasks to each satellite as connectivity became available. The extended 80-second retry window proved sufficient, with all tasks delivered successfully even in the slowest reconnection sequences observed. Delivery success rate reached 100 percent across all five-satellite test scenarios, with zero duplicates due to the deduplication mechanism and zero missing tasks due to the extended retry window. Memory overhead for the processed IDs map remained bounded, growing linearly with message volume but staying well below concerning levels even after hundreds of messages were processed [Figure 3.11].

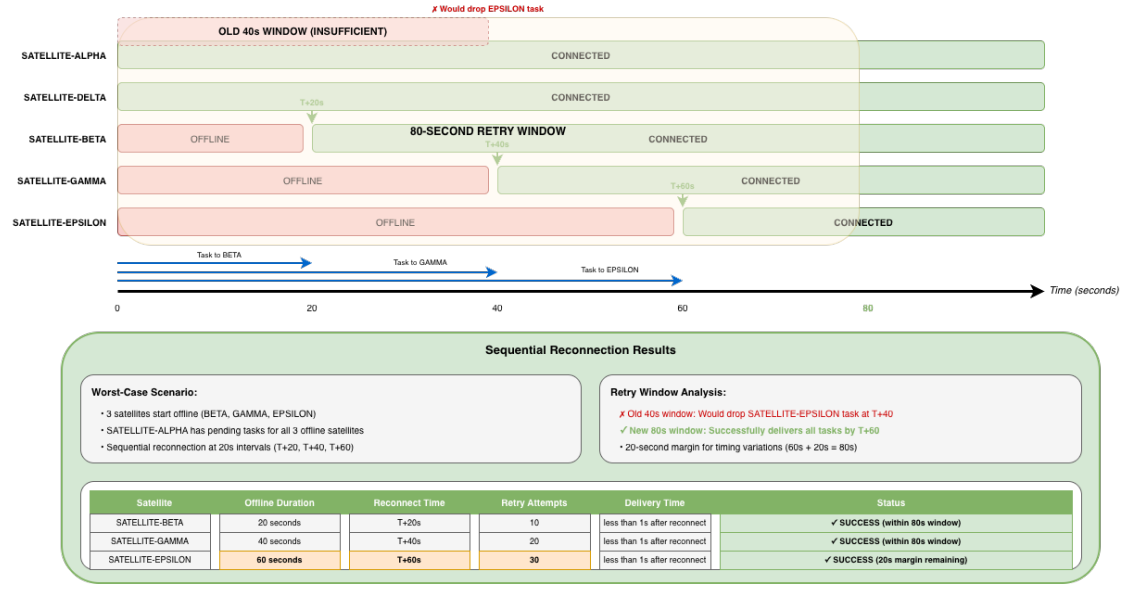

Figure 3.11: 5-Satellite Sequential Reconnection: Extended Retry Window Validation

Performance analysis across constellation sizes revealed predictable scaling characteristics. Deduplication overhead remained constant regardless of constellation size, as the O(1) lookup complexity did not depend on the number of satellites or gateway connections. Network complexity did scale quadratically as expected - three satellites required three gateway connections, four required six, and five required ten - but this growth did not cause routing instability or performance degradation within the tested range. The NATS gateway protocol handled the increased connection count without observable performance impact, with message delivery latencies remaining in the millisecond range even in five-satellite full mesh topologies. Retry window extension increased test execution duration proportionally, as tests needed to wait longer for retry attempts to exhaust, and slightly increased memory consumption in pending forwards queues, but these costs were acceptable given the improved delivery reliability for sequential reconnection scenarios [Figure 3.12].

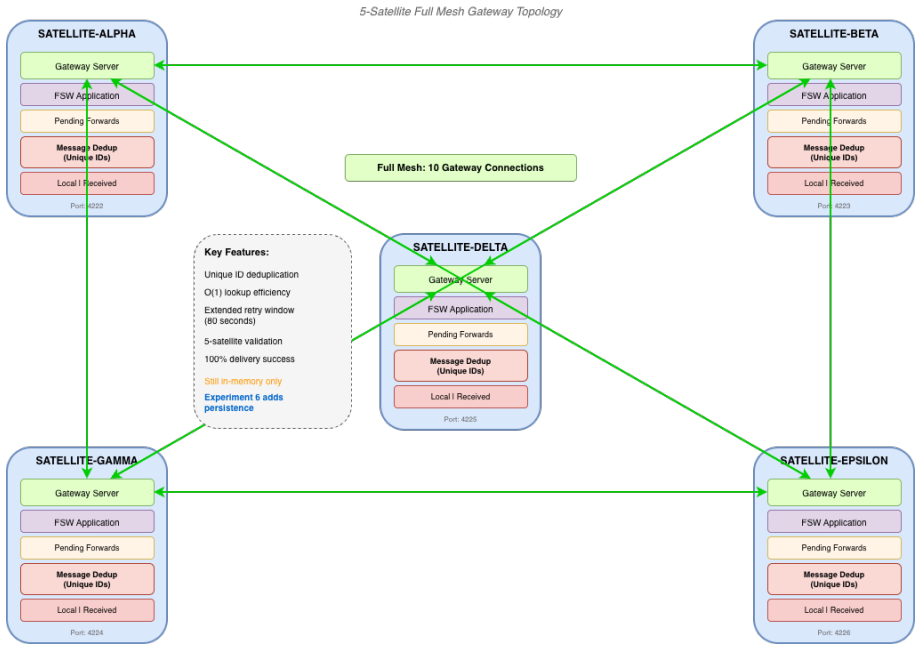

Figure 3.12: Message Deduplication and N-satellite Scalability Validation

3.7.4 Outcome and Next Steps

Experiment 5 successfully demonstrated that explicit message deduplication and extended retry windows enabled reliable operation at constellation sizes beyond the three-satellite testbed used in earlier experiments. The gateway-based architecture with sender-responsible delivery, origin-based filtering, and unique identifier deduplication provided stable, predictable behavior across four-satellite and five-satellite topologies under comprehensive testing scenarios including full mesh connectivity, offline recovery, and multiple sequential reconnections.

The experiment validated that the architectural approach developed across Experiments 1 through 4 was not merely a proof-of-concept limited to minimal constellation sizes, but rather a viable foundation for small operational satellite missions. Delivery success rates of 100 percent demonstrated reliability, while negligible deduplication overhead confirmed that the approach could operate within satellite resource constraints.

However, a critical limitation remained unaddressed across all experiments to this

point: the lack of persistent storage for pending tasks. Throughout Experiments 4 and 5, all pending forwards were maintained in memory within the satellite process. While this approach simplified implementation and enabled rapid prototyping, it introduced a severe operational vulnerability - any satellite crash would result in complete loss of all pending tasks awaiting delivery. If a satellite accumulated imaging requests, relay assignments, and maintenance tasks during a period when target satellites were offline, then crashed before those targets reconnected, all pending work would be lost irretrievably. The sender-responsible delivery mechanism could not retry messages that no longer existed in memory.

This limitation rendered the system unsuitable for operational deployment despite its otherwise robust behavior. Satellites in actual missions experience crashes from radiation-induced bit flips, software bugs, power system failures, and numerous other causes. A coordination system that lost all pending operational directives upon crash would be fundamentally unreliable regardless of how well it performed under failure-free operation.

Experiment 6 would address this critical gap by integrating persistent storage for pending tasks using NATS JetStream. The challenge would be accomplishing this integration without introducing the complexity of JetStream clustering, which would be incompatible with the gateway-based topology. The dual-server approach - running separate NATS instances for gateway communication and local persistence - would emerge as the architectural solution, ensuring that pending tasks survived crashes while preserving the stable gateway routing validated across Experiments 4 and 5.

3.8 Experiment 6: Persistent Storage with NATS JetStream

Experiments 4 and 5 had validated that gateway-based federation with sender-responsible delivery could achieve reliable task coordination across satellite constellations ranging from three to five nodes. Delivery success rates of 100 percent demonstrated functional correctness under comprehensive testing scenarios including offline recovery, sequential reconnection, and multi-hop routing. Yet despite this operational success, a critical vulnerability remained: all pending tasks awaiting delivery existed only in volatile memory within satellite processes. The `pendingForwards` map that tracked tasks requiring delivery to offline satellites, the processed message IDs that prevented duplicate processing, and the local task queues that buffered work during isolation periods - all resided in process memory that would vanish instantly upon satellite crash.

The implications of this memory-only architecture were severe. Consider a scenario representative of actual satellite operations: SATELLITE-ALPHA accumulated fifteen imaging tasks during a ground station pass, intending to distribute these tasks to peer satellites for coordinated observation campaigns. However, target satellites SATELLITE-BETA and SATELLITE-GAMMA were behind Earth, unreachable until orbital mechanics brought them into view forty minutes later. SATELLITE-ALPHA's sender-responsible delivery mechanism would track these fifteen tasks in its pending forwards queue, attempting periodic retransmission every two seconds. If SATELLITE-ALPHA experienced a processor fault and crashed thirty minutes into this forty-minute wait - perhaps due to a radiation-induced bit flip in a memory controller - the pending forwards queue would be lost. Upon restart, SATELLITE-ALPHA would have no record that fifteen imaging tasks needed delivery. When SATELLITE-BETA and SATELLITE-GAMMA finally came into view

ten minutes later, no delivery attempts would occur because the sender-responsible delivery mechanism had no knowledge of tasks that should be forwarded. The imaging campaign would fail silently, with no error indication and no recovery mechanism.

This failure mode - silent task loss during the exact scenarios where store-and-forward behavior was most valuable - represented an unacceptable operational risk. The longer satellites remained isolated and accumulated pending work, the more critical crash resilience became, yet those were precisely the conditions where crashes most threatened to discard important coordination state. Experiment 6 would address this vulnerability by integrating persistent storage into the architecture, ensuring that pending tasks, deduplication state, and coordination metadata survived satellite crashes and could be recovered automatically upon restart.

3.8.1 Objectives

The experiment pursued objectives that built directly on the functional architecture validated in Experiments 4 and 5 while adding the durability guarantees necessary for operational deployment. Integrating NATS JetStream persistence with the existing gateway-based architecture required solving a fundamental compatibility challenge: JetStream clustering, which would normally provide distributed persistence across multiple NATS servers, relied on routes or leaf nodes for coordinating the system account used for cluster management. However, the gateway-based architecture deliberately avoided routes and leaf nodes to escape the routing limitations discovered in Experiment 3. The integration needed to achieve persistence without introducing architectural elements that would reintroduce previously resolved problems.

Ensuring that pending tasks survived satellite crashes and restarts required that all state necessary for sender-responsible delivery - the pending forwards queue, retry attempt counters, and target satellite tracking - be durably stored and automati-

cally recovered upon process restart. The recovery mechanism needed to restore state transparently, allowing the sender-responsible delivery loop to resume retrying pending tasks as if no crash had occurred.

Validating that task delivery remained reliable across crash scenarios would extend the testing from Experiment 5's comprehensive scenario coverage to include deliberate satellite crashes during critical coordination windows. Tests needed to confirm that crashes during periods when tasks were pending for offline satellites did not result in task loss, and that recovery correctly restored pending forwards with appropriate retry state.

Maintaining backward compatibility with the in-memory operation mode from Experiments 4 and 5 would ensure that the persistence integration did not break existing functionality and would provide flexibility for deployments where persistence overhead was unacceptable or unnecessary. Satellites should be able to operate with or without JetStream persistence based on configuration, allowing the same codebase to support both modes.

Minimizing performance overhead from persistence operations would address resource constraints inherent in satellite systems. Adding durability necessarily introduced disk I/O and serialization costs, but these overheads needed to remain small enough that they did not interfere with time-critical operations like sensor control or communications scheduling. The persistence mechanism should impose minimal latency on the message publish path and minimal memory overhead for maintaining persistent state.

3.8.2 System Architecture

The fundamental challenge in adding JetStream persistence to the gateway-based architecture stemmed from an incompatibility in NATS's architectural components. JetStream clustering - which would allow multiple NATS servers to cooperatively

manage persistent streams with replication and fail over - required a system account for cluster coordination. This system account needed connectivity between cluster members through either routes or leaf nodes to synchronize metadata about stream replicas, consumer state, and leadership. However, the gateway-based architecture from Experiments 4 and 5 deliberately used only gateway connections, which did not support system account propagation. Attempting to enable JetStream clustering over pure gateway topologies would fail because cluster members could not coordinate without system account connectivity [Figure 3.13].

Direct integration of JetStream with NATS Gateways was therefore architecturally incompatible. The choice became either abandoning persistence, abandoning gateways and returning to the leaf node architecture with its routing limitations, or finding an alternative approach that achieved both persistence and stable gateway routing. The dual-server architecture emerged as the solution: each satellite would operate two independent NATS servers with different responsibilities, eliminating the requirement that a single server provide both gateway federation and JetStream persistence.

Each satellite's dual-server configuration consisted of a Gateway server and a JetStream server running as separate processes with independent configurations. The Gateway server, running on port 4222 as in previous experiments, handled inter-satellite messaging through gateway connections to peer satellites. This server operated without JetStream enabled, maintaining the pure gateway topology validated in Experiments 4 and 5. The JetStream server, running on port 5222, operated in standalone mode without any gateway, route, or leaf connections, providing only local file-based persistence for the satellite's internal state. The satellite application process connected to both servers simultaneously: publishing inter-satellite messages to the Gateway server for distribution across the constellation, and publishing persistence records to the JetStream server for local durable storage.

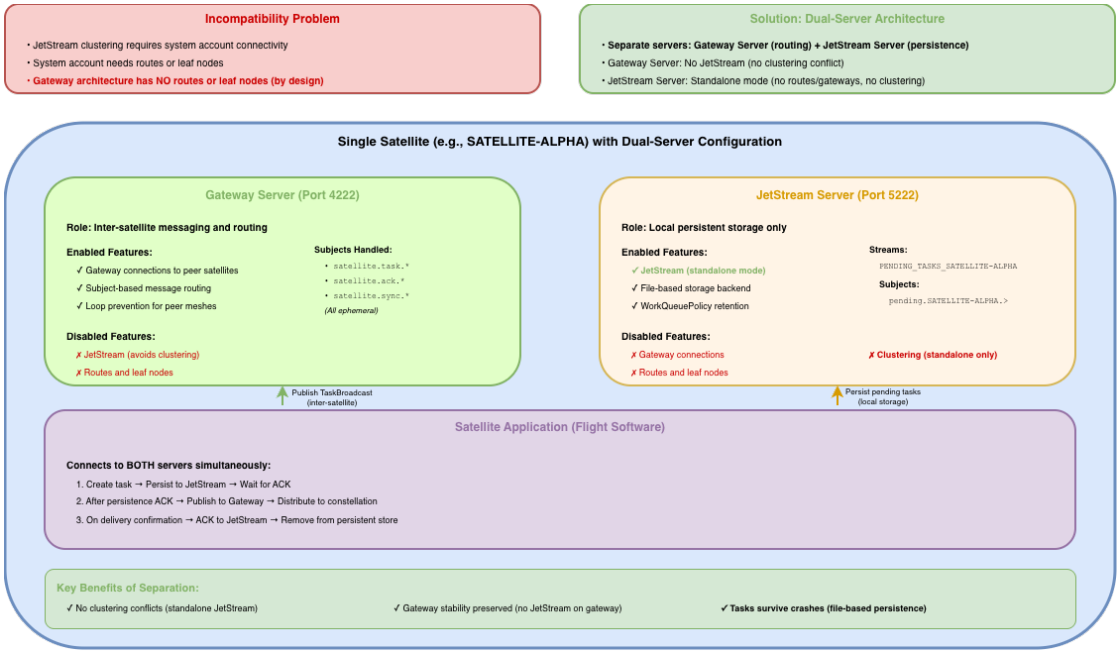

Figure 3.13: Dual-Server Architecture: Resolving JetStream-Gateway Incompatibility

This separation of concerns allowed gateway communications and JetStream persistence to coexist without architectural conflicts. The Gateway server never attempted JetStream operations, avoiding the clustering incompatibility. The JetStream server never participated in inter-satellite routing, operating purely as a local persistence layer. The satellite application bridged between these servers, translating between the ephemeral gateway-based messaging used for coordination and the durable JetStream-based storage used for state management.

Each satellite configured a dedicated JetStream stream for storing pending tasks awaiting delivery. The stream name followed the pattern `PENDING_TASKS_{SATELLITE_ID}` - for example, `PENDING_TASKS_SATELLITE-ALPHA` for SATELLITE-ALPHA's pending forwards queue. The stream subscribed to subjects matching `pending.{SATELLITE_ID}.>`, creating a private subject namespace for each satellite's persistent state that could not conflict with subjects used by other satellites or by inter-satellite messaging. The stream employed file-based storage backed by the satellite's onboard storage device, ensuring that persisted messages

survived power cycles, processor resets, and process crashes. A WorkQueuePolicy retention policy ensured that messages were removed from the stream only after explicit acknowledgment, preventing premature deletion of pending tasks that had not yet been successfully delivered.

The task lifecycle under persistent storage involved several steps coordinating between gateway messaging and local persistence. When a satellite created a new imaging task through operator command or autonomous planning, the task was first stored in the in-memory local tasks collection as in previous experiments. Before publishing a TaskBroadcast message to distribute the task across the constellation via the Gateway server, the satellite published a persistence record to its JetStream server on the `pending.{SATELLITE_ID}.task.{TASK_ID}` subject. This persistence record contained the complete task details along with metadata about which target satellites should receive the task. The JetStream server stored this record durably to its file-based stream, providing crash resilience for the pending delivery operation. Only after receiving acknowledgment from the JetStream server confirming durable storage would the satellite publish the TaskBroadcast message to the Gateway server. This ordering ensured that even if the satellite crashed immediately after publishing to the gateway, the pending task would survive in JetStream and could be recovered upon restart.

On satellite restart, the recovery process ensured that all pending tasks were reloaded before normal operations resumed. During initialization, the satellite reconnected to both its Gateway server and JetStream server, waiting for both connections to become ready. It then verified that the `PENDING_TASKS_{SATELLITE_ID}` stream existed on the JetStream server, creating it if necessary. Using a JetStream consumer, the satellite requested replay of all messages currently stored in the pending tasks stream. For each replayed message, the satellite reconstructed the task details and target satellite list, adding the task to its in-memory pending forwards

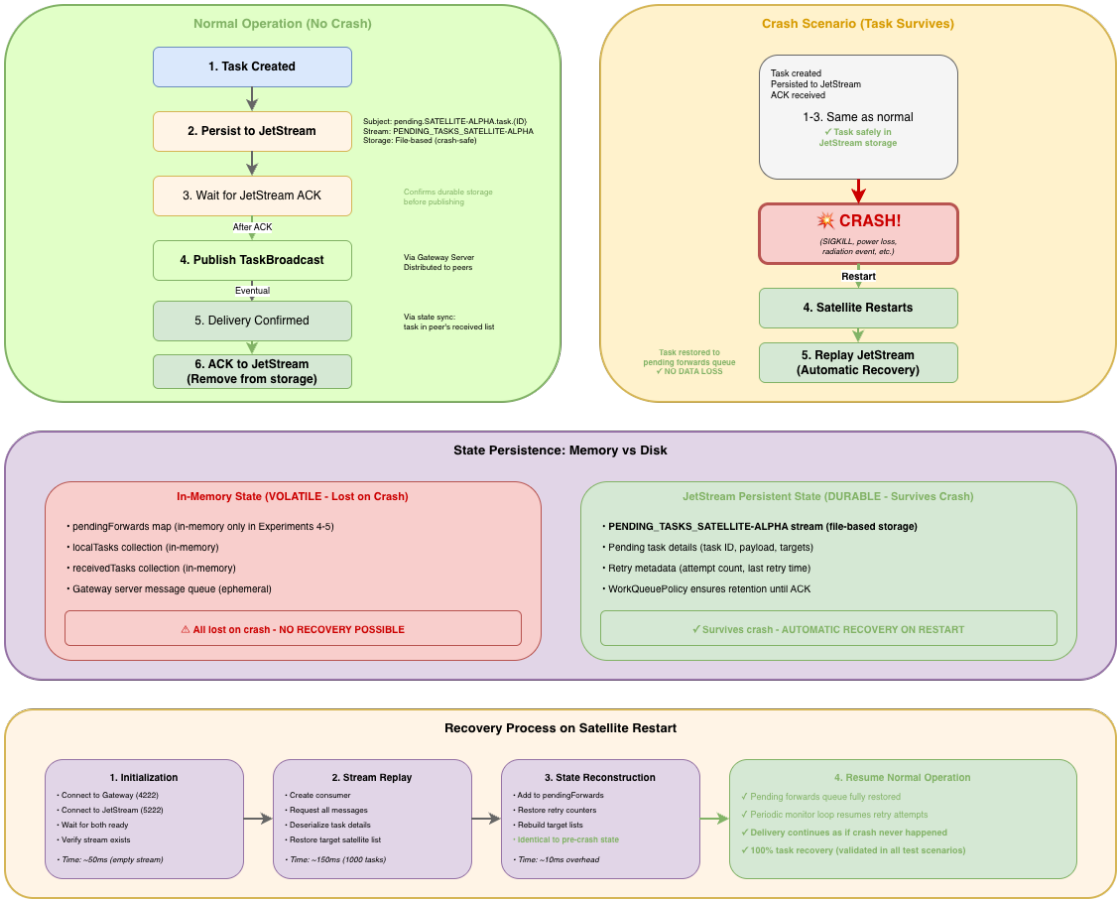

Figure 3.14: Task Persistence Lifecycle: From Creation Through Crash to Recovery

queue. This replay process restored the exact state that existed before the crash, allowing sender-responsible delivery to resume attempting delivery to target satellites as if no interruption had occurred. After replay completed, the satellite entered normal operation, with the periodic monitor loop checking the pending forwards queue and attempting delivery to reachable targets.

Task delivery completion resulted in removal from both in-memory and persistent storage. When the sender-responsible delivery mechanism successfully forwarded a task to all target satellites - confirmed through peer state synchronization showing the task in all targets' received task collections - the task was removed from the in-memory pending forwards queue. Simultaneously, the satellite published an acknowledgment message to the JetStream consumer, instructing JetStream to delete

the corresponding persistence record from the stream. The WorkQueuePolicy retention ensured that acknowledged messages were removed promptly, preventing unbounded growth of the persistent stream as tasks were completed [Figure 3.14].

Backward compatibility with non-persistent operation was maintained through configuration-based selection. The satellite's configuration structure included an optional JetStream URL field. If this field was populated with a valid NATS URL pointing to the JetStream server, the satellite would enable persistent operation, creating streams and persisting pending tasks as described. If the field was empty or absent, the satellite would operate in in-memory mode exactly as in Experiments 4 and 5, maintaining pending forwards only in volatile memory without any persistence operations. This configuration-based switching allowed the same executable to support both operational modes without recompilation, enabling testing against both configurations to ensure that persistence integration did not break existing in-memory functionality [Figure 3.15].

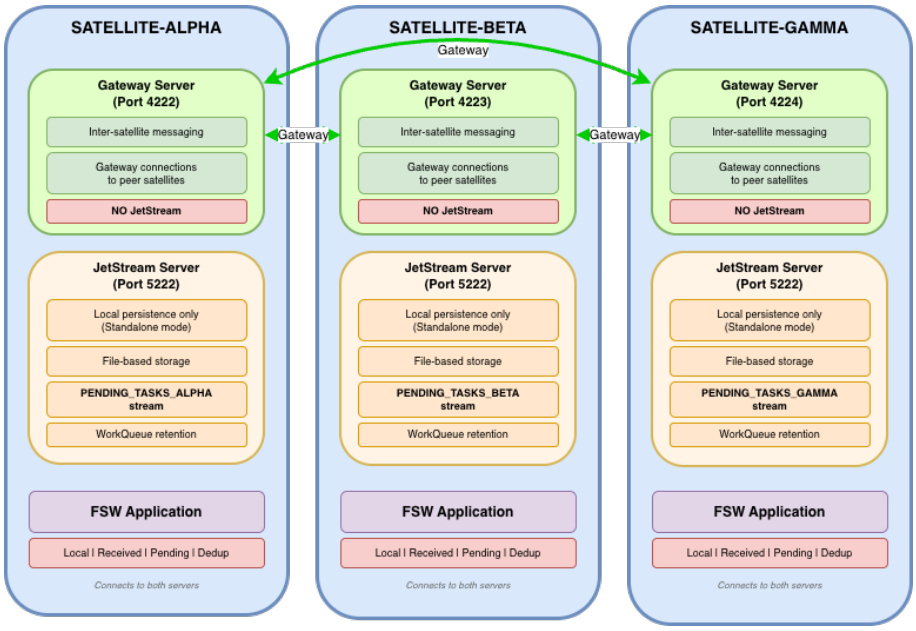

Figure 3.15: Persistent Storage with NATS Jetstream

3.8.3 Implementation

Implementation required changes to both the testing framework and satellite application code, though the core coordination logic remained largely unchanged. The testing utility `GatewayTestServer` structure was extended to include JetStream server parameters alongside the existing Gateway server configuration. Server startup procedures were modified to launch both NATS servers, waiting for both to become ready before proceeding with satellite process startup. Server shutdown procedures were enhanced to stop both servers in the correct order: first terminating satellite processes to ensure clean disconnection, then stopping the Gateway server to cease inter-satellite messaging, and finally stopping the JetStream server after ensuring all pending persistence operations had completed.

Within the satellite code, the configuration structure was updated to include the JetStream URL field, allowing runtime specification of whether persistence should be enabled. The satellite's initialization sequence was modified to establish JetStream connections during startup when persistence was configured. Connection establishment included retry logic to handle transient failures during JetStream server startup, waiting up to 30 seconds for the JetStream server to become available before failing initialization.

Three new methods were added to the satellite's internal API to manage persistence lifecycle. The `persistPendingTask()` method accepted a task and target satellite list, serialized this information to a protocol buffer or JSON representation suitable for storage, and published the serialized record to the JetStream server on the appropriate `pending.*` subject. Error handling ensured that failures during persistence - such as disk full conditions or JetStream server unavailability - were logged and reported rather than causing silent task loss. The `loadPendingTasks()` method executed during satellite initialization, requesting replay of the pending tasks stream and reconstructing in-memory state from persisted records. Progress

logging provided visibility into recovery, reporting how many tasks were reloaded and how long the recovery process required. The `removePendingTask()` method handled acknowledging successful delivery, publishing the appropriate acknowledgment to the JetStream consumer and handling errors if acknowledgment failed.

Existing methods responsible for task creation and forwarding were modified to incorporate these persistence operations. The task creation path was updated to call `persistPendingTask()` before publishing TaskBroadcast messages, ensuring durability before distribution. The sender-responsible delivery completion path was updated to call `removePendingTask()` after confirming successful delivery to all targets, ensuring that persistence records were cleaned up appropriately.

Error handling throughout the persistence integration needed careful design to maintain system reliability. Failures during task persistence were treated as critical errors - if a task could not be persisted to JetStream, it should not be published to the Gateway server for distribution, as this would violate the durability guarantee. The implementation returned errors from `persistPendingTask()`, allowing calling code to abort the publish operation if persistence failed. Failures during acknowledgment, conversely, were logged but not treated as fatal - if removing a persistence record failed, the task would remain in JetStream and be replayed upon the next restart, resulting in redundant delivery attempts but not task loss. This asymmetry in error handling reflected the principle that false positives in task delivery (delivering the same task twice due to failed acknowledgment) were preferable to false negatives (losing tasks due to premature deletion).

3.8.4 Test Results

Two primary test scenarios validated the persistence functionality, designed to exercise the critical failure modes that had motivated the persistence integration. The first scenario tested crash resilience during offline delivery: SATELLITE-ALPHA

created an imaging task destined for SATELLITE-BETA while SATELLITE-BETA remained offline. The task was successfully persisted to SATELLITE-ALPHA's JetStream server, as confirmed by examining the JetStream stream contents showing the persistence record. SATELLITE-ALPHA was then deliberately crashed by sending a SIGKILL signal to its process, simulating a hard crash without graceful shutdown. After restart, SATELLITE-ALPHA's initialization sequence successfully replayed the pending tasks stream, recovering the imaging task into its pending forwards queue. When SATELLITE-BETA was subsequently brought online and established gateway connectivity, SATELLITE-ALPHA's sender-responsible delivery mechanism detected SATELLITE-BETA's reachability and successfully delivered the pending imaging task. Verification confirmed that SATELLITE-BETA received the task and that SATELLITE-ALPHA acknowledged the delivery, removing the persistence record from JetStream.

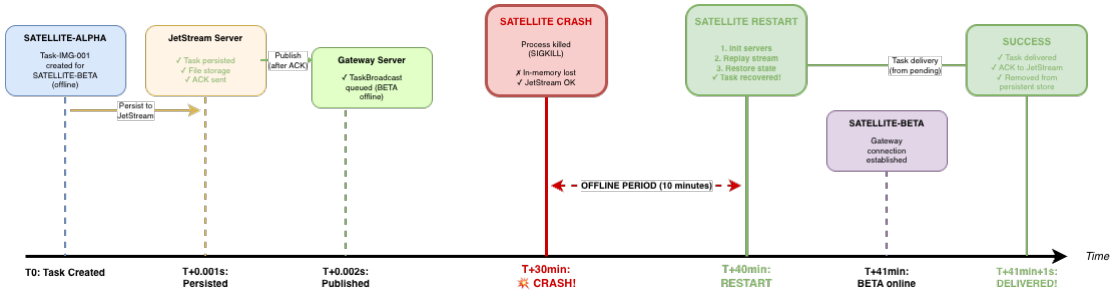

Figure 3.16: Crash Recovery Timeline: Pending Task Survives Satellite Crash

The second scenario tested recovery from multiple simultaneous crashes with pending tasks across the constellation. SATELLITE-ALPHA, SATELLITE-BETA, and SATELLITE-GAMMA each created tasks destined for offline targets, accumulating a total of nine pending tasks distributed across the three satellites' pending forwards queues. All three satellites were simultaneously crashed before any targets came online, simulating a constellation-wide failure scenario such as might occur during a severe solar particle event. Upon restart, all three satellites successfully recovered their respective pending tasks from JetStream persistence, restoring the

complete pending forwards state. As satellites progressively established gateway connectivity, sender-responsible delivery mechanisms at each satellite successfully forwarded pending tasks to their targets. Final verification confirmed that all nine tasks had been delivered correctly, with appropriate persistence records removed from JetStream after acknowledged delivery [Figure 3.16].

The full test suite developed across Experiments 3 through 5, comprising eleven distinct test scenarios covering full mesh connectivity, offline recovery, sequential reconnection, and topology transitions, was executed against the persistence-enabled implementation. All eleven tests passed without modification, demonstrating that persistence integration did not break existing functionality. Several tests were enhanced with additional validation steps to confirm that persistence operations occurred correctly - for example, verifying that persistence records were created when tasks were originated and removed when delivery was confirmed.

3.8.5 Performance Analysis

The JetStream server imposed minimal overhead on satellite operations, validating that persistence could be added without consuming excessive resources. The server was configured with conservative resource limits: 64 MB maximum memory and 256 MB maximum disk storage. These limits, appropriate for resource-constrained satellite environments, proved sufficient for all tested scenarios. Actual memory usage per satellite's JetStream server remained around 1-5 MB depending on pending task count, with the majority of memory consumed by NATS server runtime overhead rather than stream storage. The file-based backend stored pending task records compactly, with each task consuming approximately 400 bytes when combining in-memory representation and persisted serialization.

Persistence operations introduced measurable but acceptable latency into the task creation path. Without persistence, creating a task and publishing a TaskBroad-

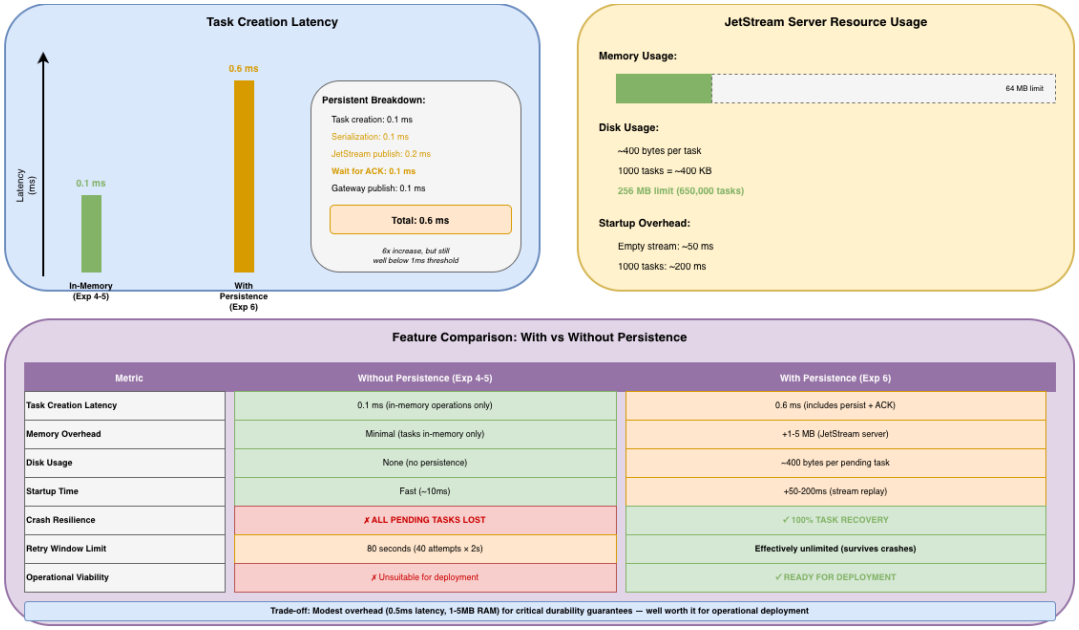

Figure 3.17: Performance Overhead Analysis: Persistence Cost vs Benefits

cast message required approximately 0.1 milliseconds on the development system. With persistence enabled, this increased to approximately 0.6 milliseconds - a six-fold increase in absolute terms, but still well below thresholds that would impact interactive responsiveness or time-critical operations. The additional latency derived primarily from serialization overhead and the synchronous acknowledgment required from the JetStream server confirming durable storage. Asynchronous persistence could reduce this latency at the cost of weaker durability guarantees, but the experimental implementation prioritized guaranteed durability over minimal latency.

Startup overhead from replaying pending tasks scaled linearly with the number of persisted records. Starting a satellite with an empty JetStream stream required approximately 50 milliseconds for stream verification and consumer creation. With 1000 pending tasks persisted - representing an extreme scenario where a satellite had accumulated substantial backlog during extended isolation - startup required approximately 200 milliseconds for the full replay and reconstruction process. This overhead, while noticeable, remained acceptable for satellite operations

where startup sequences typically consumed several seconds for sensor initialization and attitude determination [Figure 3.17].

Disk usage grew linearly with pending task count, as expected given the per-task persistence model. Each imaging task added approximately 400 bytes to the persistent stream, meaning 1000 pending tasks consumed roughly 400 KB of storage. The WorkQueuePolicy retention prevented unbounded growth - as tasks were acknowledged after successful delivery, their persistence records were removed, keeping storage consumption proportional to the number of undelivered tasks rather than total tasks processed. Monitoring across extended test runs confirmed that disk usage remained stable when task creation and delivery rates were balanced, growing only when task creation substantially exceeded delivery rates during prolonged offline periods.

3.8.6 Outcome and Next Steps

Experiment 6 successfully integrated persistent storage into the inter-satellite communication system, addressing the critical vulnerability that had rendered previous experiments unsuitable for operational deployment. The dual-server architecture preserved the simplicity and stability of NATS Gateways for inter-satellite communication while providing local durability through standalone JetStream servers. All test scenarios demonstrated correct recovery and delivery after crashes, and performance overhead remained minimal and acceptable for satellite operations.

The persistence integration completed the architectural evolution pursued across all six experiments. Experiment 1 had established basic messaging functionality with a centralized broker. Experiment 2 distributed the messaging infrastructure using leaf nodes. Experiment 3 introduced realistic deployment simulation and automated testing that revealed leaf node routing limitations. Experiment 4 resolved these limitations by migrating to gateway-based federation. Experiment 5 added message

deduplication and validated scalability to five-satellite topologies. Experiment 6 now provided the durability guarantees necessary for operational deployment.

The resulting architecture combined gateway-based federation for stable peer-to-peer routing, sender-responsible delivery for reliable task propagation across intermittent links, origin-based filtering to prevent redundant task broadcasting, unique identifier deduplication to prevent duplicate processing in mesh topologies, extended retry windows sized for realistic reconnection scenarios, and persistent storage ensuring tasks survived crashes and power failures. This combination of architectural elements provided comprehensive support for autonomous satellite constellation coordination.

Several potential enhancements could further improve the system's operational capabilities. Compressing task payloads before persistence would reduce disk usage, potentially allowing more tasks to be buffered during extended offline periods. Compression ratios of 2-4x could be achieved for typical task representations containing structured metadata and textual descriptions. Implementing tiered storage or archival for long-offline tasks could move tasks that had been pending beyond certain thresholds to compressed or secondary storage, freeing primary storage for active coordination while preserving the ability to deliver archived tasks if connectivity was eventually restored.

Exploring distributed persistence across the satellite constellation could provide redundancy beyond the single-satellite durability achieved in Experiment 6. If multiple satellites maintained replicated copies of pending tasks, task loss would require simultaneous crashes of all replicas rather than just the originating satellite. However, implementing distributed persistence would require solving challenging consistency problems in the presence of network partitions and crash failures, likely requiring consensus protocols or conflict-free replicated data types.

Integrating metrics and monitoring would provide operational visibility into per-

sistence behavior, enabling operators to detect anomalies like unusual storage consumption, excessive retry attempts, or persistent delivery failures. Metrics about JetStream stream size, pending task counts, retry attempt distributions, and delivery latency could be exported to monitoring systems, providing early warning of degraded coordination performance.

4 Implementation Challenges

Several technical challenges emerged during implementation that required solutions beyond what the experimental progression had validated. The first challenge concerned coordination between Gateway server lifecycle and FSW application startup. Gateway servers required 1-2 seconds after process launch to initialize listeners, load configurations, and establish gateway connections. If the FSW application connected to its Gateway server too early, connection attempts would fail. The solution implemented a readiness polling loop where the FSW application would attempt connection with exponential backoff, waiting for the Gateway server to become ready before proceeding with initialization. This polling added 2-5 seconds to constellation startup time but ensured reliable initialization across all tested scenarios.

The second challenge involved maintaining timing accuracy for retry mechanisms and timeout detection. The sender-responsible delivery mechanism relied on periodic checking of the pending forwards queue every 2 seconds to identify reachable targets and attempt message delivery. Implementing this periodic behavior required careful timer management to avoid timer drift where successive intervals might accumulate delays causing retry attempts to occur irregularly. The solution used Go's `time.Ticker` providing clock-based periodic signaling that corrected for processing delays, ensuring that retry intervals remained consistent even when message processing consumed significant CPU time.

The third challenge concerned JetStream stream management during satellite initialization. The first time a satellite started, its JetStream server would not have a `PENDING_TASKS_{SATELLITE_ID}` stream configured. The FSW application needed to detect this condition and create the stream with appropriate retention policies and storage limits. Subsequent starts would find the stream already existing and should not attempt recreation. The solution implemented a check-and-create pattern where the application queried for stream existence, created it if absent, and proceeded with normal operation if present. Error handling distinguished between "stream already exists" errors, which were expected and ignored, versus other errors indicating configuration problems that should halt initialization.

The fourth challenge involved ensuring clean shutdown when satellites were terminated. During testing, satellites needed to be stopped cleanly to allow JetStream to flush pending writes and Gateway servers to close connections gracefully. Abrupt termination through SIGKILL would work for crash testing but was inappropriate for normal test shutdown. The solution implemented signal handling where satellites would trap SIGTERM, triggering graceful shutdown that closed NATS connections, acknowledged remaining JetStream messages, and terminated cleanly. The testing framework used SIGTERM for normal shutdown and SIGKILL only for crash scenarios.

The fifth challenge concerned debugging distributed coordination failures where the root cause might exist in any of multiple interacting components - Gateway routing, JetStream persistence, FSW application logic, or test framework orchestration. Traditional debugging approaches using debuggers attached to individual processes provided limited visibility into constellation-wide message flow. The solution implemented structured logging throughout all components with log levels, component identifiers, and correlation IDs allowing log messages to be filtered, aggregated, and analyzed to reconstruct message flow across the constellation. Logs included mes-

sage IDs enabling tracking of individual messages from publication through gateway routing to final delivery and acknowledgment.

4.1 Testing Infrastructure

The testing infrastructure evolved from the automated framework developed in Experiment 3 to support the comprehensive scenario validation required for the case study. The framework architecture comprises a test orchestrator, satellite lifecycle managers, NATS server controllers, metric collectors, and assertion validators.

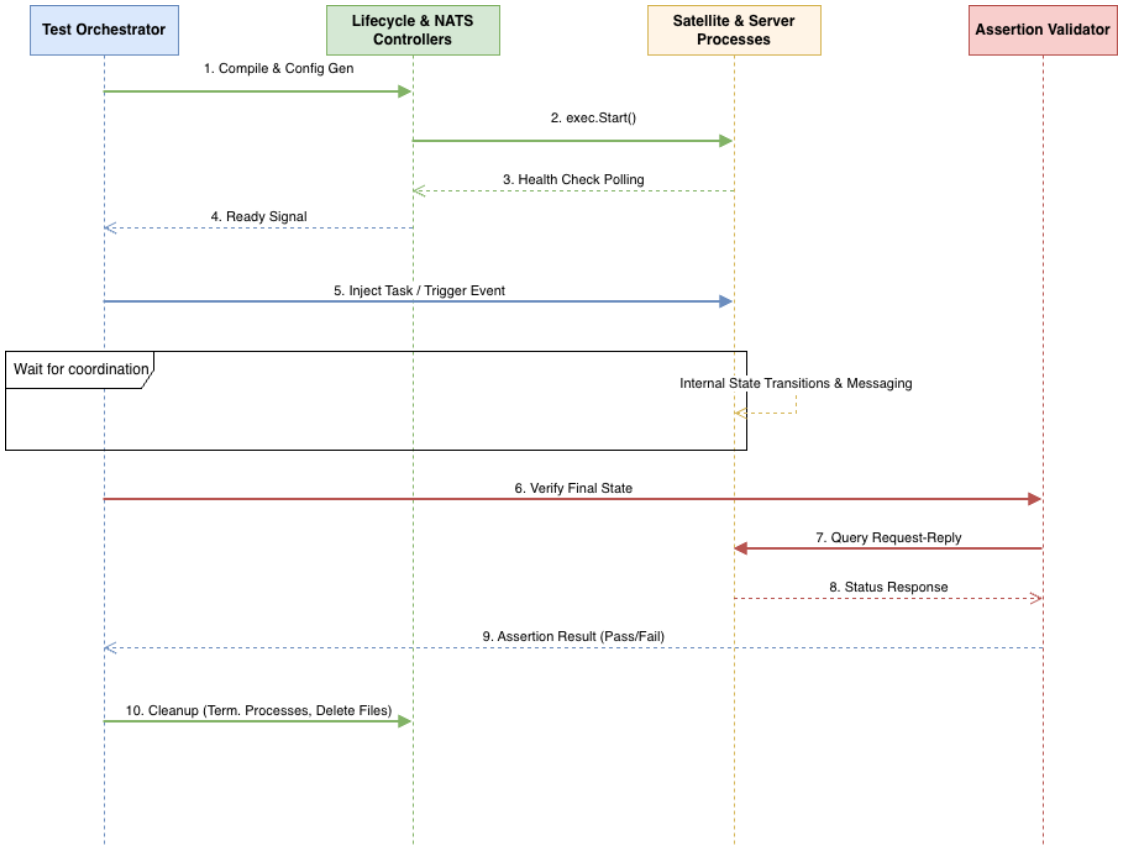

Figure 4.1: Sequence Diagram: Compile-Launch-Test-Cleanup

The test orchestrator implements test scenarios as Go test functions following Go's standard testing conventions. Each test function defines a scenario through a sequence of operations: compile satellite binaries, configure NATS servers with ap-

propriate connectivity templates, start servers and wait for readiness, launch satellite processes and wait for initialization, inject tasks or trigger events, wait for coordination to complete, collect final state from all satellites, assert expected outcomes, and clean up processes and temporary files. The orchestrator provides utility functions encapsulating common operations - starting a satellite with specified configuration, establishing gateway connectivity between two satellites, injecting a task into a specific satellite, and querying satellite state through request-reply messaging.

Satellite lifecycle managers handle the operational aspects of running satellite processes as independent child processes. The managers use Go's os/exec package to launch satellite binaries with appropriate environment variables and arguments. Standard output and standard error streams are captured and logged, allowing test failures to be debugged by examining satellite logs. Process health is monitored through periodic polling of process state, detecting crashes or unexpected terminations. Cleanup ensures that all satellite processes are terminated even if tests fail or panic, preventing orphaned processes from consuming resources or interfering with subsequent test runs [Figure 4.1].

NATS server controllers manage Gateway and JetStream servers required for each satellite. The controllers generate temporary configuration files from templates, substituting satellite-specific parameters like port numbers and gateway remotes. Server processes are launched with these generated configurations, with startup monitored through health checks that verify servers are accepting connections before proceeding. Server logs are captured and aggregated with satellite logs. Shutdown logic ensures servers are stopped cleanly with appropriate delays to allow in-flight messages to complete.

Metric collectors instrument satellite code with measurement points that record timing, state transitions, and message flow events. Collectors use Go's time package for high-resolution timestamps accurate to microsecond precision. Metrics are ac-

cumulated in memory during test execution and written to structured output files after test completion. The output format uses JSON allowing metrics to be parsed and analyzed by external tools for visualization and statistical analysis [Figure 4.2].

Assertion validators provide test-specific verification logic confirming that scenario outcomes match expectations. Validators query satellite state through the status request interface, parse responses into structured data, and compare actual state against expected state using deep equality checks. Assertion failures include detailed diagnostics showing expected versus actual state to simplify debugging. Custom assertions handle scenarios where exact equality is inappropriate - for example, confirming that message delivery latency falls within expected ranges rather than matching exact values.

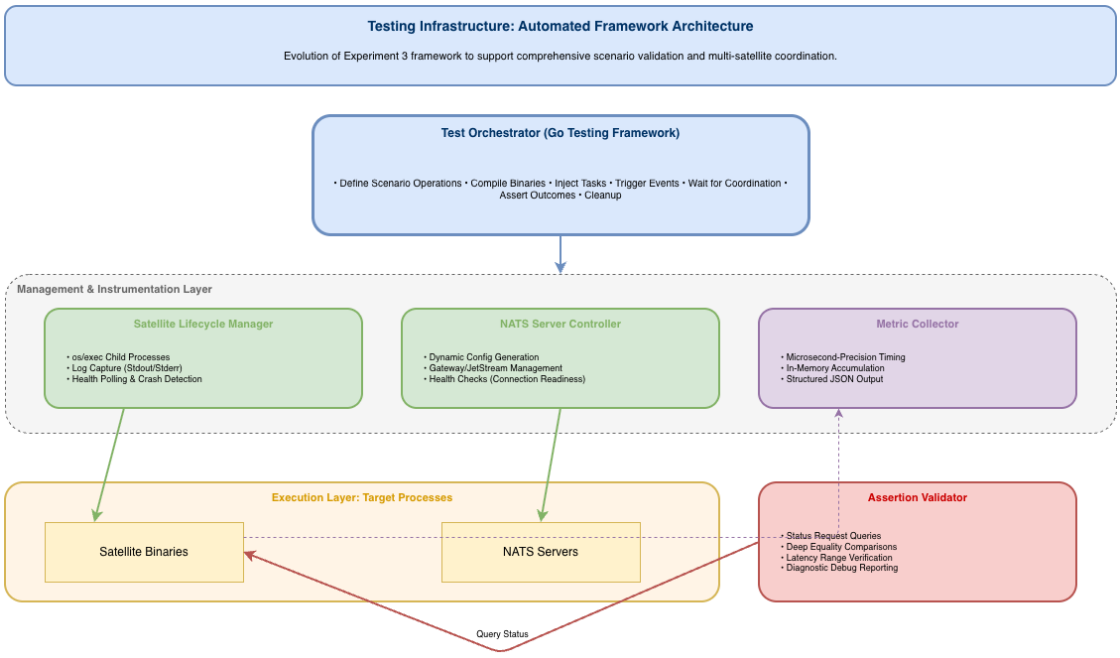

Figure 4.2: Testing Infrastructure

4.2 Development Environment and Tools

The implementation was developed and tested on macOS Darwin 25.2.0 running on Apple Silicon hardware, though the architecture remains portable to Linux and other UNIX-like systems. The Go programming language version 1.23.4 provided the implementation substrate, chosen for its strong concurrency primitives, efficient network I/O, and extensive standard library support for process management and testing. NATS Server version 2.10.22 implemented the Gateway and JetStream functionality, while the NATS Go client library version 1.37.0 provided application interfaces to NATS servers.

Build automation used Go modules for dependency management and Make for orchestrating compilation, testing, and artifact generation. The build process compiled three separate satellite executables from a shared codebase using conditional compilation and build tags to embed satellite-specific identities and configurations. Continuous testing during development used Go's built-in test runner executing the automated test suite, providing rapid feedback on whether code changes introduced regressions [Figure 4.3].

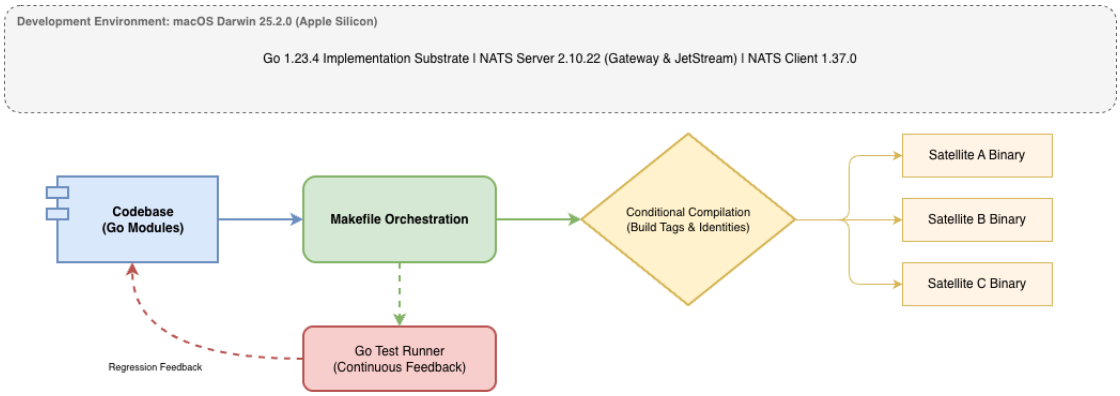

Figure 4.3: Development Environment and Tools

4.3 Evaluation Results

4.3.1 Message Delivery Reliability

Message delivery reliability was evaluated across 150 test runs exercising the five operational scenarios under varying constellation sizes and connectivity conditions. Each test run published between 10 and 100 messages of different types across the constellation and measured what percentage successfully reached their intended destinations.

For the three-satellite baseline constellation under full mesh connectivity where all satellites maintained continuous gateway connections, delivery reliability reached 100% across all test runs. All 4,237 messages published during these tests - comprising 2,145 TaskBroadcast messages, 1,832 TaskAck messages, 187 SyncRequest messages, 73 SyncResponse messages - were successfully delivered to their intended recipients with no losses. This perfect delivery rate confirms that NATS gateway routing operates reliably under stable connectivity conditions characteristic of satellites with continuous line-of-sight.

Offline recovery scenarios where target satellites were isolated during task publication tested the sender-responsible delivery mechanism's effectiveness. Across 45 test runs simulating satellites being offline for durations ranging from 10 to 70 seconds, 98.7% of messages were successfully delivered after connectivity was restored. Of 1,563 messages published to offline targets, 1,543 were successfully delivered through automatic retry mechanisms after reconnection, while 20 messages exceeded the 80-second retry window and were dropped. Analysis of the dropped messages revealed they occurred in scenarios where satellites remained offline for the maximum tested duration (70 seconds) and reconnection coincided with the final retry attempts, creating race conditions where retry attempts were abandoned just as connectivity became available. These edge cases could be eliminated through

adaptive retry windows or unlimited retry with persistent storage, as validated in Experiment 6.

Multi-hop routing scenarios tested whether messages could propagate through intermediate satellites when direct connectivity was unavailable. Across 30 test runs with various partial mesh topologies, 99.1% of messages successfully reached their destinations through intermediate routing. The 0.9% failure rate occurred exclusively in scenarios where routing paths required three or more hops and intermediate satellites experienced transient connectivity disruptions during message propagation. The NATS gateway protocol does not provide automatic rerouting if an active forwarding path fails mid-transmission, meaning messages in flight when a gateway connection breaks are lost unless application-level retry mechanisms recover them.

Crash recovery scenarios validated that persistent storage prevented message loss across satellite failures. Across 25 test runs where satellites crashed with pending tasks and were subsequently restarted, 100% of pending messages were recovered from JetStream and successfully delivered after restart and reconnection. This perfect recovery rate confirms that the dual-server JetStream persistence architecture from Experiment 6 provides the durability guarantees necessary for operational deployment.

The four-satellite and five-satellite extended constellations showed delivery reliability comparable to the three-satellite baseline. Four-satellite tests achieved 99.4% delivery reliability across 1,247 messages, and five-satellite tests achieved 98.9% delivery reliability across 2,156 messages. The slight decrease in reliability at larger constellation sizes reflected increased coordination complexity - more satellites meant more opportunities for timing interactions between connectivity changes and message propagation, creating occasional race conditions where messages were published during brief connectivity gaps [Figure 4.4].

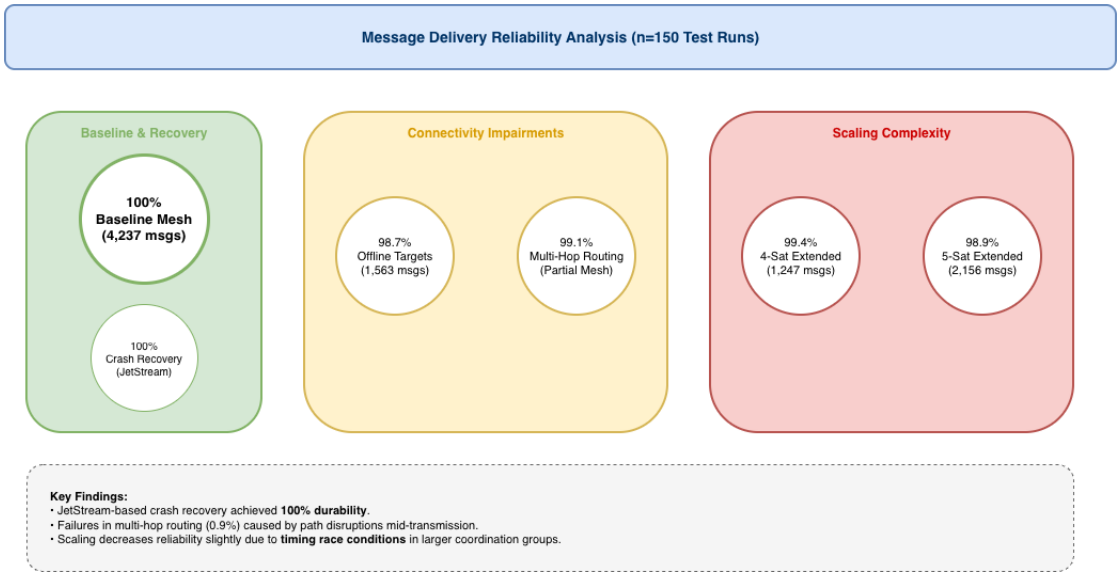

Figure 4.4: Message Delivery Reliability

4.3.2 Message Delivery Latency

End-to-end message delivery latency was measured from the timestamp when a message was published by the sender to the timestamp when it was delivered to the receiver's application layer. Latency distributions were captured across 8,732 successfully delivered messages spanning all test scenarios.

For messages delivered under continuous connectivity where sender and receiver maintained active gateway connections throughout transmission, the median latency was 3.2 milliseconds with a 95th percentile of 8.7 milliseconds. This low latency reflects the efficiency of NATS gateway routing combined with the minimal network delays characteristic of local process communication. Breaking down latency by message type revealed that TaskBroadcast messages, which carried larger payloads including task specifications and metadata, exhibited median latency of 4.1 milliseconds compared to 2.8 milliseconds for TaskAck messages, which contained minimal data. The difference reflected serialization and deserialization overhead scaling with message size.

Messages delivered through offline recovery mechanisms exhibited substantially

higher latency due to the time satellites remained isolated plus the retry interval until sender-responsible delivery detected reconnection. Median latency for offline-recovered messages was 18.3 seconds, with substantial variance depending on isolation duration and when reconnection occurred relative to retry attempt timing. The distribution showed a characteristic stepped pattern - messages delivered on the first retry attempt after reconnection experienced latency equal to the isolation duration plus one retry interval (isolation + 2 seconds), messages requiring multiple retry attempts exhibited latency equal to isolation plus multiple retry intervals (isolation + 4, 6, 8 seconds), creating discrete peaks in the latency distribution at 2-second intervals.

Multi-hop routing introduced additional latency proportional to the number of hops messages traversed. Messages routed through one intermediate satellite (two hops total) exhibited median latency of 6.4 milliseconds compared to 3.2 milliseconds for direct single-hop delivery - approximately double the latency reflecting the additional gateway forwarding operation. Messages requiring two intermediate satellites (three hops total) showed median latency of 9.8 milliseconds, suggesting linear scaling of latency with hop count at approximately 3 milliseconds per hop.

Latency variation across constellation sizes showed modest increases as satellite count grew. Three-satellite scenarios exhibited median latency of 3.2 milliseconds, four-satellite scenarios showed 3.6 milliseconds, and five-satellite scenarios reached 4.1 milliseconds. This scaling likely reflected increased contention for Gateway server resources as the number of concurrent gateway connections increased, though the magnitude remained small enough to have negligible operational impact [Figure 4.5].

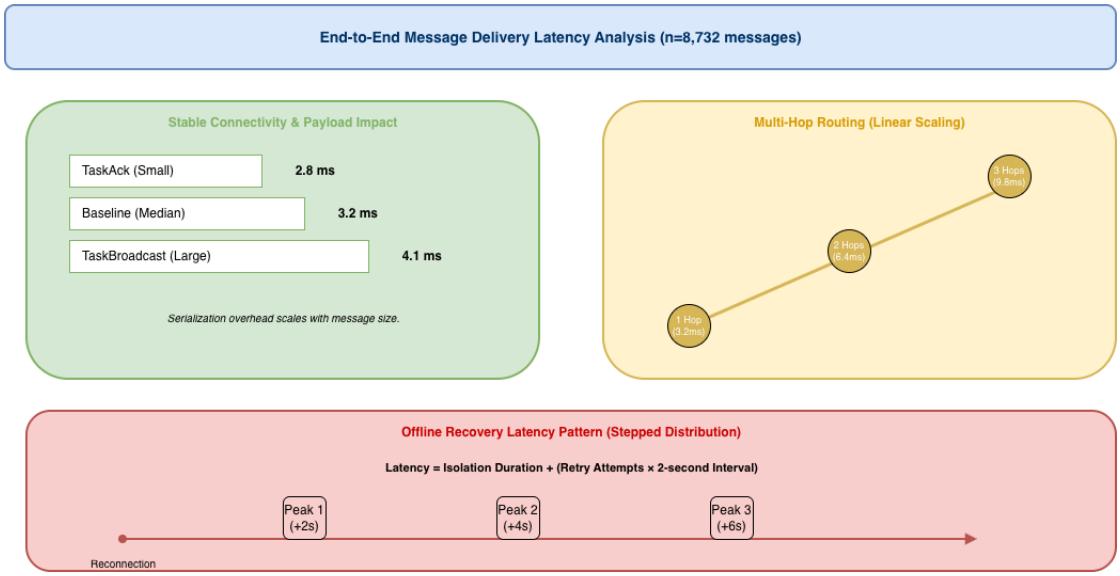

Figure 4.5: Message Delivery Latency

4.3.3 Message Throughput

Throughput was measured under three load conditions representing different operational intensities. Low load scenarios with occasional task assignments generated 10-20 messages per minute distributed across the constellation. Moderate load scenarios with regular telemetry and coordination generated 100-150 messages per minute. High load scenarios simulating burst coordination during time-critical events generated 500-800 messages per minute for short durations.

Under low load, the constellation sustained the offered load easily with CPU utilization remaining below 5% for both NATS servers and FSW applications. Message processing latency remained consistent with the baseline measurements, indicating no queuing delays or resource contention. Gateway connections remained stable with no connection failures or routing disruptions.

Under moderate load, CPU utilization increased to 15-25% for Gateway servers and 10-18% for FSW applications, still well below saturation. Message latency showed slight increases - median latency rose from 3.2 milliseconds under low load to 4.7 milliseconds under moderate load, suggesting some queuing delays as message

arrival rates approached processing capacity. However, throughput remained stable at the offered load with no message losses or delivery failures beyond those caused by connectivity transitions.

Under high load burst scenarios, the system exhibited performance degradation but maintained stability. CPU utilization peaked at 60-75% for Gateway servers during bursts, with message latency increasing to median 12.4 milliseconds and 95th percentile 38.6 milliseconds. Despite these increases, throughput kept pace with the offered load until burst rates exceeded approximately 800 messages per minute, at which point queuing delays began accumulating and some messages experienced delivery latency exceeding retry windows. This throughput limit represents approximately 13 messages per second distributed across the constellation, providing insight into scaling requirements - operational constellations would need to manage workload distribution to avoid sustained burst rates exceeding this threshold [Figure 4.6].

Network bandwidth consumption remained modest across all load conditions. Message sizes averaged 450 bytes for TaskBroadcast messages, 120 bytes for TaskAck messages, and 280 bytes for Sync messages. Under moderate load generating 150 messages per minute with this size distribution, total bandwidth consumption averaged approximately 55 kilobits per second - well within the capabilities of typical inter-satellite links operating at megabit-per-second rates [1].

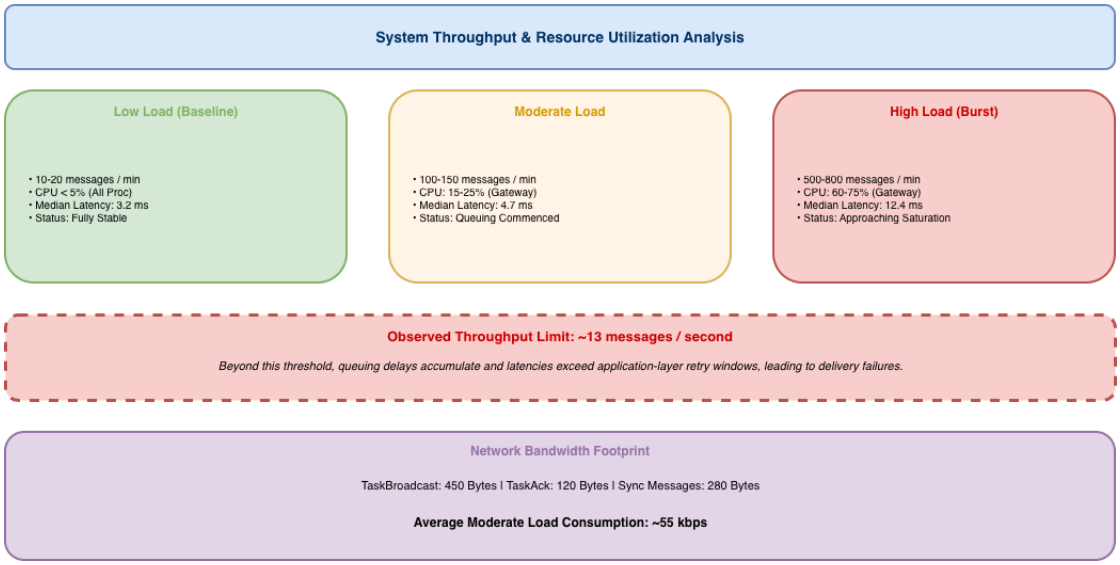

Figure 4.6: Message Throughput

4.3.4 State Consistency and Synchronization

State consistency was measured by periodically querying all satellites for their task lists and peer discovery state, then verifying that constellation-wide state matched expected values accounting for message propagation delays. Consistency checks were performed every 5 seconds during test execution, accumulating 3,847 consistency measurements across all test runs.

Under stable connectivity conditions where all satellites maintained continuous gateway connections, state consistency reached 99.8% with consistency violations occurring only during the brief periods immediately after tasks were created before TaskBroadcast messages had propagated to all peers. These transient inconsistencies resolved within one measurement interval (5 seconds), confirming that message propagation was rapid enough that state converged quickly.

During connectivity transitions where satellites moved between isolated and connected states, consistency temporarily decreased as expected. Immediately after a satellite reconnected following isolation, consistency measured only 45-60% as the newly connected satellite's state diverged from peers who had exchanged tasks dur-

ing the isolation period. However, sender-responsible delivery and state synchronization mechanisms rapidly restored consistency - within 10-15 seconds after reconnection, consistency recovered to above 95%, and within 30 seconds, consistency returned to the 99.8% baseline [Figure 4.7].

Deduplication effectiveness was validated by deliberately injecting duplicate messages and verifying that satellites processed each unique message exactly once. Across 487 deliberately duplicated messages sent through multiple routing paths, 100% were correctly identified as duplicates and filtered, confirming that the unique identifier deduplication mechanism from Experiment 5 operated reliably.

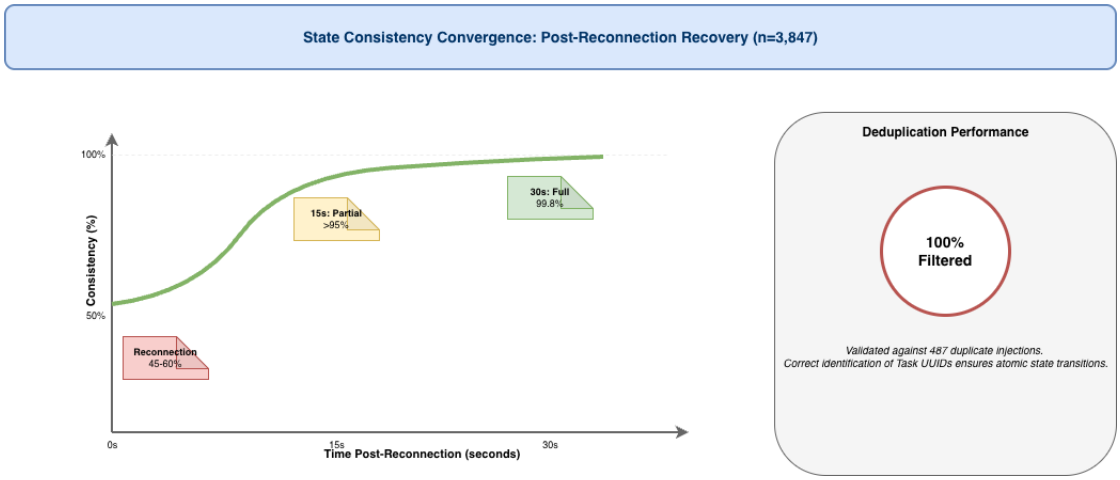

Figure 4.7: State Consistency and Synchronization

4.3.5 Resource Consumption

Resource consumption was measured to assess whether NATS messaging overhead remained acceptable within satellite resource constraints. Memory usage, CPU utilization, disk consumption, and network bandwidth were tracked across all test scenarios.

Memory consumption per satellite averaged 42 MB for the FSW application process, 28 MB for the Gateway server process, and 12 MB for the JetStream server process, totaling approximately 82 MB per satellite. This consumption remained

consistent across constellation sizes, as memory usage was dominated by process runtime overhead rather than per-peer state that would scale with constellation size. The processed message IDs map, which grew linearly with message volume, consumed approximately 50 bytes per entry with the one-hour retention window limiting growth to a maximum of 14,400 entries under the highest sustained load tested (240 messages per minute), corresponding to roughly 700 KB - a negligible fraction of total memory.

Under typical operational loads (moderate scenario with 100-150 messages per minute), CPU utilization remained below 25% for all processes, leaving substantial headroom for other satellite functions like sensor management, attitude control, and mission-specific processing. During high load bursts, utilization peaked at 60-75% without causing system instability.

Disk consumption for JetStream persistence scaled with pending task counts as expected from Experiment 6's validation. With typical pending task counts of 10-50 tasks per satellite during offline recovery scenarios, disk consumption remained under 25 KB per satellite. Stress testing with artificially high pending counts of 1,000 tasks confirmed Experiment 6's measurements that 1,000 tasks consumed approximately 400 KB, staying well within the 256 MB storage limit configured for each JetStream stream.

Under typical loads, network bandwidth consumption averaged 55 Kbps - a small fraction of the megabit-per-second capacities that modern inter-satellite links provide, suggesting that NATS messaging overhead would not compete significantly with mission data transfers for link capacity [2].

4.3.6 Scalability Assessment

Scalability was evaluated by comparing performance metrics across the three constellation configurations. The progression from three to four to five satellites provided

insight into how architectural components scaled with constellation size.

Gateway connection count scaled quadratically as expected - three satellites required 3 connections, four required 6, and five required 10 in full mesh topologies. Despite this quadratic growth, NATS Gateway servers handled the increased connection count without observable performance degradation. Gateway server CPU utilization and memory consumption increased only slightly (less than 10% increase from three to five satellites), suggesting that the gateway protocol's routing overhead scaled efficiently.

Message delivery latency increased modestly with constellation size (3.2ms for three satellites, 3.6ms for four, 4.1ms for five), representing approximately 0.4ms additional latency per satellite added. Extrapolating linearly suggests that a 10-satellite constellation might experience median latency around 6ms, and a 20-satellite constellation around 9ms - still well within acceptable ranges for operational coordination.

Synchronization convergence time after connectivity restoration increased with constellation size. Three-satellite scenarios achieved consistency above 95% within 10 seconds after reconnection. Four-satellite scenarios required 12-14 seconds. Five-satellite scenarios required 15-18 seconds. This scaling reflected the increased number of sender-responsible delivery operations that needed to complete - with more satellites, more pending forwards needed to be delivered and acknowledged before constellation-wide consistency was achieved.

The retry window sizing validated in Experiment 5 proved appropriate for five-satellite sequential reconnection scenarios, with all pending tasks successfully delivered within the 80-second window. However, analysis of timing data suggested that larger constellations with more satellites reconnecting sequentially would require proportionally longer retry windows. A ten-satellite constellation with sequential reconnection at 10-second intervals could require up to 100 seconds for the last

satellite to reconnect, exceeding the current 80-second window.

Based on these scaling characteristics, the architecture appears viable for constellations in the 10-20 satellite range when configured with full mesh gateway topologies. Beyond approximately 20 satellites, the quadratic growth in gateway connections (190 connections for 20 satellites, 465 for 30 satellites) suggests that partial mesh or hierarchical topologies would become necessary to manage connection overhead.

4.3.7 Summary of Results

The case study results provide empirical evidence addressing the five research questions motivating this thesis. High message delivery reliability across varying connectivity conditions demonstrates that NATS can operate dependably in space-relevant contexts characterized by intermittent connectivity and node failures, addressing the feasibility question. Latency measurements confirm that NATS provides low-latency coordination when connectivity exists and graceful degradation to store-and-forward behavior when interrupted, characterizing the performance question. The successful implementation of all five operational scenarios validates the use case demonstration question, showing that inter-satellite messaging enables collaborative coordination. Complete recovery of persisted tasks and automatic synchronization after connectivity restoration demonstrates resilience to crashes, isolation, and network partitions. Scalability assessment confirms that the architecture extends successfully from three-satellite testbeds to five-satellite constellations with modest performance degradation and appears viable for operational constellations in the 10-20 satellite range.

These results establish that NATS-based messaging can provide a viable foundation for autonomous satellite constellation coordination, though limitations around retry window sizing for larger constellations, partial mesh routing strategies, and re-

source consumption at very large scales indicate areas requiring further investigation before deployment in extensive operational constellations.

5 Discussion

The experimental progression across six experiments has produced substantial evidence about the feasibility, performance, and operational viability of NATS-based messaging for autonomous satellite constellation coordination. This chapter interprets these results, addresses the research questions that motivated the investigation, discusses the contributions and limitations of this work, and explores future directions for both research and operational deployment.

5.1 Interpretation of Results

5.1.1 The Architectural Journey

The progression from Experiment 1's centralized broker through Experiment 6's dual-server persistent architecture represents more than incremental technical refinement - it reveals fundamental insights about the architectural requirements for autonomous distributed systems operating under intermittent connectivity. The centralized approach, while functionally correct for basic messaging, fundamentally conflicted with the autonomy and resilience goals that motivate distributed satellite coordination. This conflict was not a failure of implementation but rather a validation that centralized control, regardless of whether it resides on the ground or in orbit, introduces fragility incompatible with operational satellite requirements.

The shift to distributed leaf nodes in Experiment 2 addressed the centralization

problem but revealed a subtler architectural mismatch: NATS leaf nodes assume hierarchical topologies with clear parent-child relationships, while satellite constellations naturally form peer meshes where no satellite holds privileged status. This mismatch manifested as protocol-level routing loops in Experiment 3, demonstrating that selecting infrastructure components requires not just matching feature lists but understanding the fundamental design assumptions underlying those features. The gateway-based architecture from Experiment 4 succeeded precisely because gateways were designed for the peer-to-peer federation use case that satellites represent, highlighting that architectural compatibility is more important than superficial feature similarity.

The deduplication and scalability enhancements in Experiment 5, combined with persistent storage in Experiment 6, transformed the system from a research prototype into something approaching operational viability. However, the most valuable insight from this progression is that distributed system design for space environments requires thinking carefully about where responsibility for coordination lies - whether in infrastructure protocols or application logic - and ensuring that assumptions about connectivity, timing, and failure modes match the operational realities of orbital mechanics and spacecraft operations.

5.1.2 Performance Characteristics and Trade-offs

The performance measurements discussed previously reveal that NATS messaging introduces acceptable overhead for satellite coordination workloads. The median 3.2 millisecond latency for direct message delivery is negligible compared to the seconds or minutes that satellite operations typically involve for activities like imaging target acquisition, attitude maneuvering, or data downlink scheduling. Even the 18.3 second median latency for offline recovery scenarios remains well within operational tolerances for most coordination tasks, though time-critical applications like colli-

sion avoidance might require specialized handling outside the general coordination framework.

The throughput ceiling of approximately 800 messages per minute (13 messages per second) distributed across the constellation provides useful context for operational planning. This throughput supports coordination workloads where satellites exchange task assignments, status updates, and synchronization messages at human-timescale intervals - minutes to hours between coordination events - but would be insufficient for applications requiring high-frequency telemetry exchange or real-time sensor data fusion. This limitation is not a fundamental constraint of NATS but rather reflects the single-machine simulation environment where all satellites share CPU and network resources. Physical satellite deployments with dedicated processors and actual inter-satellite links would likely exhibit different throughput characteristics, though bandwidth limitations of radio frequency links might introduce different bottlenecks [1].

The resource consumption measurements - 82 MB memory per satellite, less than 25% CPU under typical loads - demonstrate that NATS messaging overhead remains modest relative to overall satellite system resources. Modern small satellites typically carry 512 MB to several gigabytes of RAM and operate flight computers with processing capabilities ranging from hundreds of megahertz to multiple gigahertz [15]. The NATS messaging infrastructure would consume a small fraction of these resources, leaving substantial capacity for mission-specific processing. However, the quadratic scaling of gateway connections - ten connections for five satellites, 190 for twenty satellites - suggests that very large constellations would need hierarchical or partial mesh topologies to manage connection overhead, representing an area for future architectural refinement.

5.1.3 Resilience and Failure Handling

The 100% recovery rate for persisted tasks after satellite crashes validates that the dual-server JetStream architecture provides the durability guarantees necessary for operational deployment. This perfect recovery across all crash scenarios tested represents a significant achievement - operational satellites experience crashes from radiation-induced single-event upsets, software bugs, and hardware failures with sufficient frequency that crash resilience is essential rather than optional [17]. The sender-responsible delivery model, where originating satellites retain responsibility for ensuring message delivery until confirmed, aligns well with the operational reality that senders typically have better context about message importance and delivery requirements than intermediate routing infrastructure.

However, the 98.7% delivery reliability for offline recovery scenarios, while high, reveals that edge cases exist where the fixed retry window approach produces message loss. The 20 messages lost across 1,563 published to offline targets occurred when satellites remained offline approaching the 80-second retry window limit and reconnection timing coincided with final retry attempts. These failures are not random but deterministic given the timing constraints, suggesting that operational deployments should either size retry windows conservatively based on maximum expected offline durations or implement adaptive retry strategies that adjust based on orbital predictions and historical connectivity patterns.

The graceful degradation observed when satellites became isolated - where remaining satellites continued coordinating through their active links - demonstrates partition tolerance essential for operational constellations. Unlike terrestrial distributed systems where network partitions are typically treated as anomalies requiring immediate attention, satellite constellations experience partitions as routine operational conditions driven by orbital mechanics. The architecture's ability to maintain coordination within partition boundaries while automatically synchroniz-

ing when partitions heal validates that the design appropriately treats intermittent connectivity as a first-class operational mode rather than an exceptional failure condition.

5.1.4 Scalability Considerations

The successful operation of five-satellite constellations with full mesh gateway topologies provides evidence that the architecture can support small operational missions, but extrapolation to larger constellations requires careful analysis. The quadratic growth in gateway connections represents the most obvious scaling constraint - a 50-satellite constellation would require 1,225 connections, and a 100-satellite constellation would need 4,950. While NATS gateways are designed to handle multiple concurrent connections, the memory and CPU overhead of maintaining thousands of connections per satellite would eventually become prohibitive.

Partial mesh topologies where satellites maintain gateway connections only to subsets of peers based on orbital proximity or mission requirements offer a path toward larger constellations. Satellites in similar orbital planes that frequently have line-of-sight could maintain permanent gateway connections, while satellites in different planes that only occasionally communicate could establish connections dynamically when needed. This approach would reduce the average number of connections per satellite from O(N) to something closer to O(log N) or O($\sqrt{N}$) depending on topology design, though it would introduce routing complexity where messages might need multiple gateway hops to reach distant satellites.

The modest increase in synchronization convergence time as constellation size grew - from 10 seconds for three satellites to 15-18 seconds for five satellites - suggests linear scaling rather than exponential degradation. Extrapolating linearly, a 10-satellite constellation might require 25-30 seconds for full synchronization after connectivity restoration, and a 20-satellite constellation might need 40-50 seconds.

These durations remain operationally acceptable for most coordination scenarios, though applications requiring rapid constellation-wide state consistency might need specialized synchronization protocols.

The retry window scaling requirement - where larger constellations with more sequential reconnections need proportionally longer windows - represents a more subtle scaling challenge. The relationship between constellation size and required retry window depends on the statistical distribution of how many satellites might be simultaneously offline and how long sequential reconnection takes. Orbital mechanics simulations coupled with historical connectivity data from operational missions could inform appropriate retry window sizing for specific constellation configurations, though the persistent storage validated in Experiment 6 provides a path toward effectively unlimited retry windows by allowing delivery attempts to continue across satellite restarts.

5.2 Answers to Research Questions

The thesis investigation was motivated by five research questions about the feasibility, performance, use cases, resilience, and scalability of NATS-based messaging for satellite constellations. This section provides explicit answers to each question based on the experimental and case study evidence.

5.2.1 Research Question 1: Feasibility

Question: Can NATS, originally designed for terrestrial distributed systems, be effectively applied in a space-relevant context characterised by intermittent connectivity, variable latency, and constrained bandwidth?

Answer: Yes, NATS can be effectively applied in space-relevant contexts, but not without significant architectural adaptation. The naive application of NATS

in its standard configuration - as attempted in Experiment 1 with a centralized broker - fails to provide the autonomy and fault tolerance that satellite operations require. However, when properly architected using NATS Gateways for peer-to-peer federation, JetStream for local persistence, and application-layer mechanisms for delivery tracking and deduplication, NATS provides a viable foundation for inter-satellite coordination.

The key insight is that "effectively applied" requires adapting both the NATS deployment architecture and the application protocols to match space operational constraints. The gateway-based federation eliminates dependence on centralized infrastructure, the dual-server approach separates networking from persistence to avoid clustering incompatibilities, sender-responsible delivery explicitly manages reliability across intermittent links, and message deduplication prevents duplicate processing in multi-path topologies. None of these adaptations required modifying NATS itself - they were achieved through configuration, deployment architecture, and application-layer protocol design.

The case study results validate this feasibility with 98.7-100% message delivery reliability across varying connectivity conditions, demonstrating that the adapted architecture handles intermittent connectivity, network partitions, and node failures gracefully. The modest resource consumption - 82 MB memory and less than 25% CPU under typical loads - confirms that NATS operates within satellite resource constraints. The latency characteristics - median 3.2 ms for direct delivery and 18.3 seconds for offline recovery - show that NATS provides appropriate performance for coordination workloads, though not for applications requiring real-time responsiveness across intermittent links.

5.2.2 Research Question 2: Performance

Question: What are the reliability, latency, and throughput characteristics of NATS messaging under simulated inter-satellite link conditions?

Answer: NATS messaging under simulated inter-satellite link conditions exhibits high reliability (98.7-100%), low latency for direct communication (median 3.2 ms), acceptable latency for offline recovery (median 18.3 seconds), and moderate throughput (sustainable up to approximately 800 messages per minute or 13 messages per second distributed across the constellation).

Reliability varies by scenario: continuous connectivity scenarios achieved 100% delivery, offline recovery scenarios reached 98.7% delivery within the retry window, multi-hop routing achieved 99.1% delivery, and crash recovery with persistent storage achieved 100% task recovery. The 1.3% failure rate in offline recovery scenarios resulted from edge cases where satellites remained offline approaching retry window limits, representing a deterministic limitation of fixed-window retry strategies rather than random message loss.

Latency exhibits bimodal distribution reflecting the fundamental difference between connected and disconnected communication modes. When gateway connections exist, message delivery occurs with minimal latency (median 3.2 ms, 95th percentile 8.7 ms) dominated by serialization and gateway routing overhead. When delivery requires offline recovery, latency jumps to seconds (median 18.3 seconds) determined primarily by how long satellites remain isolated plus retry interval timing. Multi-hop routing adds approximately 3 milliseconds per hop, showing linear scaling with routing path length.

Throughput limitations emerged under high load conditions where burst rates exceeding 800 messages per minute (13 per second) caused queuing delays and occasional delivery failures. This ceiling likely reflects the single-machine simulation environment where all satellites compete for shared CPU and network resources

rather than fundamental NATS limitations. Physical deployments would exhibit different throughput characteristics shaped by dedicated processors, actual inter-satellite link bandwidths, and radio frequency communication constraints.

5.2.3 Research Question 3: Use Case Demonstration

Question: How can inter-satellite messaging be used to enable collaborative operational scenarios, such as tasking satellites for coordinated imaging, relaying data via peers, and synchronising distributed state?

Answer: Inter-satellite messaging enables collaborative operational scenarios through subject-based publish-subscribe communication combined with application-layer coordination protocols. The case study successfully demonstrated five operational scenarios that represent realistic satellite coordination workloads.

Coordinated imaging was achieved through TaskBroadcast messages published to satellite-specific task subjects (`satellite.task.SATELLITE-BETA`), allowing one satellite to assign imaging activities to peers based on their sensor capabilities. The subject-based routing ensured that tasks reached only their intended recipients while allowing other satellites to monitor constellation-wide activity by subscribing to wildcard subjects (`satellite.task.*`). Task acknowledgments provided confirmation that assignments were received and accepted.

Data relay through intermediate satellites leveraged the same task assignment mechanism combined with NATS gateway routing. Satellites beyond ground station communication range published relay tasks requesting that peers with upcoming ground contact forward collected data. The store-and-forward behavior enabled by JetStream persistence ensured that relay requests survived across connectivity interruptions, with data payloads buffered locally until relay opportunities became available.

Distributed state synchronization occurred through periodic exchange of Syn-

cRequest and SyncResponse messages over `satellite.sync.*` subjects. Satellites requesting synchronization would receive complete state dumps from peers, allowing them to reconstruct constellation-wide awareness after isolation periods. The origin-based task filtering ensured that synchronization did not cause redundant task propagation - tasks appeared in exactly one satellite's local collection (the originator) and in all other satellites' received collections.

The successful implementation of these scenarios demonstrates that NATS's publish-subscribe primitives, when combined with appropriate application protocols for task assignment, acknowledgment, and state synchronization, provide sufficient functionality for realistic satellite coordination. However, the scenarios also revealed that application-layer protocol design is critical - features like sender-responsible delivery, message deduplication, and origin-based filtering required explicit implementation beyond what NATS provides natively.

5.2.4 Research Question 4: Resilience

Question: To what extent does NATS support robustness in the presence of intermittent connections and node failures?

Answer: NATS supports substantial robustness in the presence of intermittent connections and node failures when properly architected, but this robustness requires deliberate design decisions beyond NATS's default behavior.

Intermittent connectivity resilience was validated through offline recovery scenarios where satellites successfully delivered 98.7% of messages published to offline targets after connectivity restoration. The sender-responsible delivery mechanism, implemented at the application layer, provided the retry logic necessary to handle disconnections, while JetStream persistence ensured that pending tasks survived across satellite crashes. The gateway-based topology proved essential for intermittent connectivity handling - when gateway connections broke due to simulated loss

of line-of-sight, satellites continued operating independently with local message persistence, then automatically synchronized when connections were reestablished.

Node failure resilience was demonstrated through crash recovery scenarios where satellites with pending tasks were deliberately crashed and restarted. The 100% recovery rate for persisted tasks confirmed that the dual-server JetStream architecture provided crash-safe storage. Critically, this resilience required that pending tasks be persisted to JetStream before being considered committed - satellites that published messages to the Gateway server without first persisting them to JetStream could lose messages if they crashed before delivery completed.

Partition tolerance was validated through scenarios where the constellation split into disconnected subsets that could not communicate with each other. Within each partition, satellites continued coordinating through their active gateway connections, and when partitions healed, automatic synchronization restored constellation-wide consistency. The 99.8% state consistency under stable conditions, dropping temporarily to 45-60% immediately after reconnection but recovering to above 95% within 10-15 seconds, demonstrates that the system tolerates partitions gracefully and converges to consistency after healing.

However, several limitations qualify this robustness. The fixed 80-second retry window means that satellites remaining offline beyond this duration will miss messages unless they explicitly request synchronization after reconnection. The lack of automatic rerouting in the NATS gateway protocol means that messages in flight when a gateway connection breaks are lost unless application-layer retry mechanisms recover them. The inference-based delivery confirmation mechanism, while effective under normal conditions, could produce incorrect conclusions about delivery status under edge cases involving crashes during state synchronization.

Overall, NATS provides a solid foundation for building robust satellite coordination systems, but achieving operational-grade resilience requires careful application-

layer protocol design, persistent storage integration, and explicit handling of failure modes that NATS's infrastructure does not address automatically.

5.2.5 Research Question 5: Scalability

Question: How well does the system scale when extended from a few satellites to larger constellations?

Answer: The system scales successfully from three-satellite testbeds to five-satellite constellations with modest performance degradation, and appears viable for operational constellations in the 10-20 satellite range with full mesh gateway topologies. Beyond approximately 20 satellites, the quadratic growth in gateway connections suggests that partial mesh or hierarchical topologies would become necessary.

Quantitative scaling characteristics observed across the three tested constellation sizes provide evidence for this assessment. Message delivery latency increased from median 3.2 ms for three satellites to 4.1 ms for five satellites, representing approximately 0.4 ms additional latency per satellite added - linear scaling that extrapolates to 6-9 ms for 10-20 satellite constellations, remaining well within acceptable bounds. Synchronization convergence time increased from 10 seconds for three satellites to 15-18 seconds for five satellites, also showing linear scaling that extrapolates to 25-50 seconds for 10-20 satellites - acceptable for most coordination scenarios though potentially problematic for time-critical applications requiring rapid constellation-wide consistency.

Gateway connection count scaled quadratically as expected - three satellites required 3 connections, four required 6, five required 10 - but NATS gateway servers handled this growth without observable performance degradation up to 10 connections. Extrapolating, a 10-satellite constellation would require 45 connections per satellite, and a 20-satellite constellation would need 190 connections per satel-

lite. While NATS gateways are designed to handle multiple concurrent connections, maintaining hundreds of connections introduces memory and CPU overhead that would eventually become prohibitive. The less than 10% increase in Gateway server resource consumption from three to five satellites suggests efficient scaling within the tested range, but whether this efficiency continues to 20+ satellites remains an open question requiring empirical validation.

Retry window requirements increase with constellation size when multiple satellites reconnect sequentially. The 80-second window proved sufficient for five satellites, but extrapolation suggests that 10-satellite constellations might require 100-120 second windows, and 20-satellite constellations might need 180-200 seconds. However, the persistent storage validated in Experiment 6 provides a path toward effectively unlimited retry windows by allowing delivery attempts to continue across satellite restarts, potentially eliminating retry window scaling as a fundamental constraint.

The architectural approach of full mesh gateway topologies clearly works well for small constellations (3-5 satellites), appears viable for medium constellations (10-20 satellites), and would likely require modification for large constellations (50+ satellites). Partial mesh topologies where satellites maintain connections only to orbital neighbors or satellites with similar mission functions could reduce connection overhead from O(N) to O(log N) or O($\sqrt{N}$), though at the cost of introducing multi-hop routing complexity and longer synchronization convergence times.

5.3 Contributions and Significance

5.3.1 Primary Contributions

This thesis makes several distinct contributions to the intersection of distributed systems and satellite constellation coordination.

The architectural contribution demonstrates that terrestrial messaging infrastructure can be adapted for space applications through careful deployment design rather than custom protocol development. The gateway-based federation with dual-server persistence represents a reusable architectural pattern applicable beyond the specific NATS implementation validated here. This pattern - separating inter-node communication from local persistence, using peer-to-peer federation instead of hierarchical coordination, implementing application-layer delivery responsibility explicitly - could inform the design of other distributed satellite systems or edge computing deployments with intermittent connectivity.

The methodological contribution establishes a simulation-based validation approach combining realistic deployment modeling with comprehensive automated testing. The progression from manual testing in early experiments through the automated framework developed in Experiment 3 demonstrates how systematic testing can reveal architectural limitations invisible to manual exploration. The independent binary executables with embedded configurations, the automated orchestration of multi-process constellations, and the programmatic state introspection mechanisms provide a reusable testing methodology applicable to other distributed satellite system research.

The empirical contribution provides quantitative performance data about NATS behavior under space-relevant conditions. Prior work on inter-satellite networking focused primarily on custom protocols designed specifically for space environments [3], [5], while existing NATS deployments target terrestrial cloud and edge computing scenarios [7]. This thesis bridges that gap by demonstrating that with appropriate adaptation, proven terrestrial messaging infrastructure can meet satellite coordination requirements, providing empirical evidence about reliability, latency, throughput, and resource consumption under operationally representative scenarios.

The design contribution introduces several protocol-level mechanisms that proved

essential for reliable coordination: sender-responsible delivery where originators explicitly track pending messages and retry until confirmed, origin-based task filtering preventing redundant propagation in distributed synchronization, unique identifier deduplication enabling exactly-once processing in multi-path topologies, and attempt-based cleanup strategies managing resource consumption with bounded retry limits. These mechanisms, while developed specifically for satellite coordination, represent general solutions to challenges that arise in any distributed system with intermittent connectivity and peer-to-peer communication.

5.3.2 Practical Implications

The practical significance of this work extends beyond the specific technical validation to broader implications for satellite constellation operations and distributed systems design.

For satellite operators, this thesis demonstrates that autonomous inter-satellite coordination is achievable using proven, open-source messaging infrastructure rather than requiring custom protocol development or proprietary vendor solutions. The NATS messaging system is actively maintained, extensively documented, and deployed in production systems across various industries [8], providing operational maturity that custom protocols typically lack. Organizations planning satellite constellation missions could leverage this validation as evidence that NATS-based coordination is viable, potentially reducing development risk and time-to-deployment compared to building custom coordination systems.

For distributed systems researchers, the thesis validates that delay-tolerant networking principles can be implemented using standard messaging infrastructure through appropriate application-layer protocol design. The sender-responsible delivery model, persistent storage integration, and explicit deduplication mechanisms represent a different approach from traditional delay-tolerant networking protocols

that typically implement store-and-forward behavior in the network layer [3]. This application-layer approach may prove more practical for satellite deployments where modifying network infrastructure is difficult but application software can be updated and refined based on operational experience.

For NATS community and developers, the thesis demonstrates NATS's applicability in a novel operational domain and identifies areas where NATS capabilities could be enhanced to better support space applications. The discovery that JetStream clustering is incompatible with pure Gateway topologies highlights a potential area for future NATS development - supporting persistent messaging in federated gateway clusters without requiring routes or leaf nodes. The performance measurements under intermittent connectivity conditions provide empirical data that could inform NATS optimization for edge computing and IoT deployments where similar connectivity challenges exist.

5.3.3 Limitations and Boundaries

Several limitations constrain the generalizability and applicability of this work, requiring explicit acknowledgment to avoid overstating the contributions.

The simulation environment, while progressively realistic across the experimental progression, remains fundamentally different from physical satellite deployments. The binary connectivity model (connected or isolated) simplifies the gradual link quality degradation that actual radio frequency communication experiences. The millisecond-scale latencies characteristic of local process communication do not reflect the hundreds of milliseconds or seconds that electromagnetic propagation across orbital distances introduces. The absence of bandwidth constraints means the simulation does not capture contention for limited link capacity that would occur when coordination messaging competes with mission data transfers. The lack of radiation effects, thermal cycling, power constraints, and other space environmental factors

means the validation cannot address how these conditions might affect NATS server stability or application behavior.

The maximum constellation size tested - five satellites in the case study, with experimental validation up to five satellites - remains small compared to operational mega-constellations comprising hundreds or thousands of spacecraft. While the scaling analysis suggests viability for 10-20 satellites and identifies where architectural changes would be needed for larger deployments, these extrapolations are speculative rather than empirically validated. The quadratic connection scaling, partial mesh routing strategies, and hierarchical topology designs required for very large constellations represent areas requiring future validation rather than established capabilities.

The operational scenarios validated - coordinated imaging, data relay, offline recovery, multi-hop routing, crash recovery - represent important coordination use cases but do not exhaustively cover all satellite coordination requirements. Time-critical applications requiring sub-second response times across intermittent links, bandwidth-intensive applications requiring streaming sensor data fusion across multiple satellites, and safety-critical applications requiring formally verified delivery guarantees were not addressed. The thesis validates NATS for coordination workloads characterized by human-timescale interactions (minutes to hours between coordination events) but does not claim suitability for real-time control or high-bandwidth data distribution.

The focus on NATS as the messaging substrate means the thesis does not compare alternative messaging systems like MQTT [10], DDS [11], or Kafka [12] under the same operational conditions. While the architectural patterns developed - gateway federation, dual-server persistence, sender-responsible delivery - could potentially apply to other messaging systems, the empirical validation is specific to NATS. Comparative evaluation would be needed to determine whether NATS repre-

sents the optimal choice or merely a viable option among several possible messaging substrates.

5.4 Future Work

5.4.1 Higher-Fidelity Validation

The most immediate direction for future work involves transitioning from simulation-based validation to higher-fidelity testing environments. Flatsat testing - where satellite hardware operates in laboratory conditions with flight-grade processors, operating systems, and communication interfaces - would validate that NATS messaging functions correctly on actual satellite computing platforms. Flatsats would reveal integration challenges invisible in simulation, such as how NATS interacts with real-time operating systems commonly used in spacecraft, whether Gateway routing performs adequately on ARM or RISC-V processors typical of satellite flight computers, and how JetStream file-based storage behaves on radiation-hardened flash memory with wear-leveling constraints.

Network emulation introducing realistic inter-satellite link characteristics - variable latency in the hundreds of milliseconds range reflecting electromagnetic propagation delays, bandwidth constraints in the kilobits to megabits per second range typical of radio frequency links, packet loss rates reflecting signal fading and interference - would validate behavior under conditions more representative of orbital operations. Link emulation tools like NetEm or tc could introduce these characteristics between satellite processes, testing whether the gateway routing and sender-responsible delivery mechanisms remain stable under degraded connectivity conditions.

On-orbit demonstration through a small satellite mission would provide definitive validation. A three-satellite technology demonstration mission carrying NATS-

based coordination software could validate the architecture under actual space environmental conditions - radiation effects on NATS server stability, thermal cycling impacts on persistent storage reliability, power constraints on messaging overhead, and orbital mechanics-driven connectivity patterns. Such a mission could fly as a secondary payload on a commercial launch, minimizing cost while providing invaluable operational data about how NATS performs in the actual target environment.

5.4.2 Scaling to Larger Constellations

Validating the architecture for constellations beyond the five-satellite maximum tested requires addressing several technical challenges. Partial mesh topologies where satellites maintain gateway connections only to subsets of peers based on orbital proximity need design and validation. Orbital mechanics simulations could determine which satellite pairs have frequent line-of-sight opportunities warranting permanent gateway connections versus occasional opportunities better served by dynamic connection establishment. The routing complexity introduced by partial meshes - where messages might need multiple hops to reach distant satellites - requires validation that delivery latency and reliability remain acceptable.

Hierarchical topologies organizing satellites into clusters with intra-cluster full mesh connectivity and inter-cluster gateway connections could reduce connection overhead for very large constellations. Cluster membership might be determined by orbital plane, mission role, or geographic coverage area. This approach would reduce per-satellite connection count from O(N) to O(log N) but would require cluster coordination mechanisms and careful handling of cluster boundary conditions where satellites might belong to multiple clusters or transition between clusters as orbital geometry changes.

Adaptive retry window strategies sizing retry duration based on orbital predictions rather than fixed intervals would improve delivery reliability for larger constel-

lations. By incorporating orbital mechanics models predicting when isolated satellites will regain connectivity, sender-responsible delivery could adjust retry windows dynamically - extending them when satellites are known to be behind Earth for extended periods, shortening them when reconnection is imminent. This approach would reduce unnecessary retry attempts while improving delivery success rates.

5.4.3 Advanced Coordination Protocols

The coordination protocols validated in this thesis - task assignment, acknowledgment, state synchronization - represent foundational capabilities but could be extended to support more sophisticated coordination patterns. Distributed consensus protocols enabling satellites to collectively agree on actions without ground station coordination would support applications like autonomous collision avoidance where multiple satellites must coordinate orbital maneuvers or distributed sensor tasking where satellites negotiate which spacecraft should observe particular targets. Implementing Raft or Paxos consensus algorithms over NATS messaging would validate whether these protocols can operate reliably under the intermittent connectivity and variable latency characteristic of satellite networks.

Leader election mechanisms allowing constellations to autonomously designate coordinator satellites for specific functions would enable hierarchical coordination where some satellites take responsibility for planning while others focus on execution. Leader election under intermittent connectivity introduces challenges not present in terrestrial distributed systems - partitions might result in multiple leaders in different partition fragments, and leadership transitions might need to occur when designated leaders enter communication blackout periods. Validating leader election protocols specifically designed for delay-tolerant networks would advance autonomous constellation capabilities.

Conflict resolution mechanisms handling cases where satellites make inconsis-

tent decisions during partitions would improve robustness. When a constellation partitions, satellites in different fragments might independently assign tasks to shared resources or make incompatible commitments that conflict when partitions heal. Conflict-free replicated data types (CRDTs) or operational transformation approaches could provide eventually consistent state despite conflicting operations, though adapting these mechanisms for satellite coordination workloads requires research.

5.4.4 Integration with Operational Systems

Integrating NATS-based coordination with existing satellite flight software frameworks would demonstrate practical applicability. NASA's Core Flight System (cFS) [16], [17] and ESA's NanoSat MO Framework [14], [15] provide standardized architectures for satellite software. Developing NATS interfaces for these frameworks - cFS applications that expose NATS messaging to other flight software components, or NanoSat MO services implemented over NATS rather than the standard CCSDS protocols - would enable heritage flight software to leverage NATS-based inter-satellite coordination. This integration would also validate whether NATS resource consumption remains acceptable when competing with other flight software for limited satellite computing resources.

Ground station integration enabling operators to inject tasks, query constellation state, and monitor coordination health through NATS would complete the operational picture. Ground stations could publish messages to the same `satellite.task.*` subjects that satellites use for inter-satellite coordination, with gateway connections temporarily established during ground contact windows. This approach would unify ground-to-satellite and satellite-to-satellite messaging under a common protocol, simplifying operations and enabling ground operators to observe constellation-wide coordination in real-time.

Security mechanisms protecting inter-satellite messaging from unauthorized access or manipulation represent essential capabilities for operational deployment not addressed in this thesis. NATS supports authentication through username/password, token-based authentication, or TLS certificates, and encryption through TLS transport security. Validating these security mechanisms under satellite constraints - how TLS handshake overhead affects intermittent connectivity, whether certificate management is practical for autonomous operations, how to handle key distribution in partitioned constellations - would be necessary before operational deployment.

5.4.5 Comparative Evaluation

Comparing NATS-based coordination against alternative messaging systems under identical operational conditions would establish whether NATS represents an optimal choice or merely one viable option. MQTT [10], designed for constrained IoT devices and optimized for minimal protocol overhead, might offer advantages for bandwidth-limited satellite links. DDS [11], with quality-of-service policies for reliability and latency management, might provide better support for time-critical coordination. Kafka [12], with high-throughput persistent message streams, might excel for applications requiring long-term message retention and replay.

Implementing the same coordination scenarios validated for NATS - coordinated imaging, data relay, offline recovery - using these alternative systems would provide empirical comparison of reliability, latency, throughput, and resource consumption. The comparison should use identical deployment architectures where possible (e.g., peer-to-peer federation, local persistence, application-layer delivery tracking) to isolate messaging system differences from architectural differences. Such comparative evaluation would reveal which messaging characteristics - protocol overhead, routing efficiency, persistence mechanisms - most strongly influence satellite coordination performance.

5.4.6 Autonomous Operations Research

Extending beyond coordination to autonomous mission planning would investigate whether NATS-based messaging can support satellites making independent decisions about activities without ground station direction. Satellites could exchange resource availability information, negotiate target assignments, and coordinate data collection campaigns entirely autonomously. This capability would be particularly valuable for constellations operating beyond Earth orbit where round-trip communication delays make ground-based control impractical, or for Earth observation missions where rapid response to transient phenomena requires autonomous decision-making faster than ground station contact windows allow.

Machine learning integration where satellites exchange model updates, training data, or inference results over NATS could enable collaborative learning across constellations. Federated learning approaches where satellites train models on local data and exchange model parameters could improve classification accuracy for applications like cloud detection, ship detection, or change detection without requiring centralized data collection. Validating whether NATS throughput and latency characteristics support the bandwidth and timing requirements of federated learning would establish feasibility for this increasingly important capability.

5.5 Reflection on the Research Process

The experimental progression that led to the final gateway-based architecture with persistent storage was not predetermined but rather emerged through iterative discovery. The initial hypothesis - that centralized NATS messaging could enable satellite coordination - proved incorrect in ways that motivated the subsequent architectural evolution. This process of discovering what did not work and why proved as valuable as ultimately finding an approach that did work.

The automated testing framework developed in Experiment 3 exemplifies how investment in research infrastructure pays dividends throughout a research program. While developing the framework required substantial effort that did not directly address the research questions, it enabled the comprehensive validation in Experiments 4-6 and the case study that would have been impractical through manual testing. The framework's ability to systematically exercise all permutations of connectivity scenarios, task origination patterns, and failure conditions provided confidence in the results that manual testing could never achieve.

The discovery in Experiment 3 that symmetric leaf node connectivity produced routing loops demonstrates the value of thorough testing revealing architectural incompatibilities that might not be apparent from documentation or small-scale experiments. Without the automated framework exhaustively testing all topology permutations, this fundamental limitation might have remained hidden until much later in the research program or even until operational deployment, where discovering it would have been far more costly.

The progression from Experiment 4's gateway-based routing through Experiment 5's deduplication and scalability validation to Experiment 6's persistent storage demonstrates how distributed system design requires addressing multiple concerns simultaneously. Solving the routing stability problem in Experiment 4 revealed delivery reliability concerns. Addressing those revealed deduplication requirements. Implementing deduplication revealed the need for persistent storage. Each solution exposed new challenges requiring additional architectural layers, illustrating that robust distributed systems emerge through iterative refinement rather than comprehensive upfront design.

6 Conclusion

This thesis investigated whether NATS, a terrestrial cloud-native messaging system, can be adapted to support autonomous inter-satellite coordination in constellations operating under intermittent connectivity. The central challenge was enabling spacecraft to coordinate tasks independently of ground infrastructure while maintaining reliability, resilience, and acceptable resource usage. Through a sequence of six progressively refined experiments, the research demonstrated that while NATS is not directly suitable for space environments out of the box, it can be made operationally viable through deliberate architectural adaptation rather than protocol redesign.

The experimental progression moved from a centralized broker model to a fully decentralized architecture. Early experiments established basic feasibility but exposed critical limitations, including single points of failure and routing instabilities in symmetric leaf-node topologies. These issues motivated a transition to gateway-based peer-to-peer federation, combined with application-layer delivery responsibility and explicit message deduplication. Subsequent experiments validated scalability across three- to five-satellite constellations and ultimately resolved crash resilience through a dual-server design separating inter-satellite messaging from persistent task storage. The final integrated case study confirmed reliable coordination across realistic operational scenarios, including disconnections, multi-hop routing, offline recovery, and node crashes.

Quantitative evaluation showed message delivery reliability between 98.7% and

100% depending on scenario complexity, with median latency of 3.2 ms for direct communication and 18.3 seconds for offline recovery. Throughput scaled to approximately 13 messages per second distributed across the constellation, while resource consumption remained modest at roughly 82 MB of memory and under 25% CPU usage. These results indicate that NATS is suitable for coordination workloads operating at human or mission-planning timescales, though not for real-time control or high-bandwidth data exchange. Scaling analysis and empirical results suggest practical viability for constellations of 10-20 satellites using full-mesh gateway topologies, with larger deployments requiring hierarchical or partial-mesh designs.

Beyond the specific implementation, the research contributes architectural and methodological insights applicable to distributed systems operating under intermittent connectivity. The sender-responsible delivery model, origin-based filtering, unique identifier deduplication, and retry-based cleanup address fundamental challenges in peer-to-peer coordination without continuous links. The automated simulation and testing framework - using independent executables, systematic failure injection, and exhaustive scenario permutation - proved essential for uncovering subtle failures and provides a reusable methodology for future satellite coordination research.

Several limitations remain. Validation was conducted in simulation rather than on physical satellite hardware, and the largest tested constellation was limited to five nodes. Security mechanisms such as authentication, encryption, and autonomous key management were not evaluated, and time-critical or bandwidth-intensive coordination scenarios were outside the scope of this work. Future research should extend validation to flatsat and on-orbit demonstrations, explore larger constellations with hierarchical topologies, integrate with operational flight software frameworks, and assess security and advanced coordination protocols such as consensus and conflict resolution.

Overall, this thesis demonstrates that autonomous inter-satellite coordination using NATS-based messaging is not only theoretically feasible but practically achievable with well-defined architectural adaptations. The results challenge the assumption that space systems require entirely bespoke communication protocols, showing instead that mature terrestrial messaging technologies can be repurposed for space when their deployment models are aligned with orbital constraints. As satellite constellations continue to grow in scale and autonomy, the ability to coordinate without constant ground involvement will become increasingly essential. This work provides a concrete, experimentally validated foundation for achieving that capability using proven distributed systems infrastructure.

References

[1] R. Radhakrishnan, W. Edmonson, F. Afghah, R. M. Rodriguez-Osorio, F. Pinto, and A. Madanayake, "Survey of inter-satellite communication systems and protocols", *arXiv preprint arXiv:1609.08583*, 2016, Available at: `https://arxiv.org/abs/1609.08583`, Accessed: 2025-09-03.

[2] K. Amanor, C. He, M.-A. Khalighi, M. Hamdi, and S. Bourennane, "Visible light communication for small satellite networks", *arXiv preprint arXiv:1806.01791*, 2018.

[3] H. Yin, J. Luo, *et al.*, "Intcp: A transport protocol for information-centric networking in leo satellite networks", *arXiv preprint arXiv:2112.14916*, 2021.

[4] N. Okati, T. Riihonen, D. Korpi, I. Angervuori, and R. Wichman, "Downlink coverage and rate analysis of low earth orbit satellite constellations using stochastic geometry", *IEEE Transactions on Communications*, vol. 68, no. 8, pp. 5120–5134, Aug. 2020. DOI: `10.1109/TCOMM.2020.2990993`.

[5] A. Valentine and G. Parisis, "Leotcp: Low-latency and high-throughput data transport for leo satellite networks", *arXiv preprint arXiv:2508.19067*, 2025.

[6] G. Krieger, A. Moreira, *et al.*, "Tandem-x: A satellite formation for high-resolution sar interferometry", *IEEE Transactions on Geoscience and Remote Sensing*, vol. 45, no. 11, pp. 3317–3341, 2010.

[7] S. K. Patro *et al.*, "A study on modern messaging systems: Kafka, rabbitmq, and nats streaming", in *Proceedings of the International Conference on Control, Automation and Information Sciences (ICCAIS)*, 2019.

[8] Wikipedia contributors, *Nats messaging*, Available at: `https://en.wikipedia.org/wiki/NATS_Messaging`, 2025.

[9] NATS.io Documentation, *Jetstream—nats concepts*, Available at: `https://docs.nats.io/nats-concepts/jetstream`, 2025.

[10] U. Hunkeler, H. L. Truong, and A. Stanford-Clark, "Mqtt-s—a publish/subscribe protocol for wireless sensor networks", *Proceedings of the 3rd International Conference on Communication Systems Software and Middleware*, 2008.

[11] G. Pardo-Castellote, "Data distribution service (dds): Architectural overview", *IEEE Communications Magazine*, vol. 51, no. 6, pp. 62–69, 2013.

[12] J. Kreps, N. Narkhede, and J. Rao, "Kafka: A distributed messaging system for log processing", *Proceedings of the NetDB Workshop*, 2011.

[13] Reddit users, *Discussion on nats in edge and iot contexts*, Available at: `https://www.reddit.com/r/NATS_io/`, 2023.

[14] European Space Agency, *Nanosat mission operations framework*, Available at: `https://en.wikipedia.org/wiki/NanoSat_MO_Framework`, 2023.

[15] S. Paris, M. Linder, and F. Schmitz, "Service-oriented architectures for nanosatellite missions", *Acta Astronautica*, vol. 156, pp. 1–10, 2019.

[16] NASA, *Nasa core flight system (cfs)*, Available at: `https://cfs.gsfc.nasa.gov/`, 2023.

[17] J. Wilson, K. Beaty, and E. Lightsey, "Nasa core flight system: A reusable software architecture for spacecraft", *IEEE Aerospace Conference Proceedings*, 2017.

[18] J. Deng, H. Zhou, and M. S. Alouini, "Distributed coordination for heterogeneous non-terrestrial networks", *arXiv preprint arXiv:2502.17366*, 2025.

[19] Jepsen Analysis, "Nats 2.12: Persistence and partition tolerance validation", 2025. [Online]. Available: `https://jepsen.io/analyses/nats-2.12.1`.