NEBULA - NETWORK-ENHANCED BUNDLES FOR UNIFIED LOAD-AWARE STORAGE
6.180 DP FINAL REPORT

TEAM LACURTS-10 (JAMIE LIM, NOAH YARED, JORGE VASQUEZ)

RECITATION INSTRUCTOR: KATRINA LACURTS
WRAP INSTRUCTOR: JESSIE STICKGOLD-SARAH

1. Introduction

Communication through space poses many challenges, such as long and variable roundtrip times, intermittent and unpredictable connectivity, as well as constrained storage and bandwidth. To combat these challenges and enhance NASA's SolarNet system for extraterrestrial communications beyond point-to-point links, this paper presents the NEBULA (Network-Enhanced Bundles for Unified Load-Aware Storage) system, which extends and specifies the existing Bundle Protocol to prioritize reliable storage and quick transmission of information across the network amidst environmental uncertainties.

The unreliable space connections in SolarNet pose serious risks, delaying critical data such as hazard warnings, spacecraft telemetry and software updates, endangering space crews and communities on Earth. NEBULA's primary objective is to ensure important data is reliably and fully transmitted to their destinations, and that this is completed in time (fulfilling time requirements in the spec).

Reliability is a crucial factor to consider due to the nature of the physical medium. Data can easily lose its meaning if too much of it is missing or incorrect. For example, if a partial software update bundle is used, updates would not be able to complete, which may expose security risks or inefficiencies; this results in the network not working as intended, which has a detrimental effect on other operations involving communication on the network.

Timeliness is also important so that space and ground crews can receive timely information needed for operations and receive sufficiently early warning notice of upcoming emergencies, such as solar flares or landslides; such information is either obsolete or puts people in unnecessary danger if received too late. Specifically, when striving for reliability, NEBULA tends to depend more on some form of data duplication rather than responses and retransmissions after timeouts, as the latter would significantly increase the time needed for data to reach their final destinations due to the long roundtrip times.

NEBULA uses various mechanisms to ensure that there is enough space to store the data in the network while preventing data loss (node-layer storage, garbage collection, distributed storage), to transmit these data for processing and storage across different nodes (queue management, bundle forwarding and fragmentation),

and to continuously verify correct transmission along the path (custody and reporting). This is achieved without compromising too much on the time it takes the data to reach their destinations.

1.1. **System Overview.** NEBULA consists of the following modules:

- **node-layer storage**, which stores the bundles read from the queue, improving database performance and filtering on each individual node
- **garbage collection**, which clears up storage space on each node to provide more available space for newer bundles, while minimizing losses from removing possibly useful data
- **distributed storage**, which ensures reliability of data storage throughout the network
- **queue management**, which determines how/which bundles are sent out when requested
- **bundle forwarding and fragmentation**, which allows data to reach their desired destinations
- **custody and reporting**, which ensures reliability of data transmission throughout the network

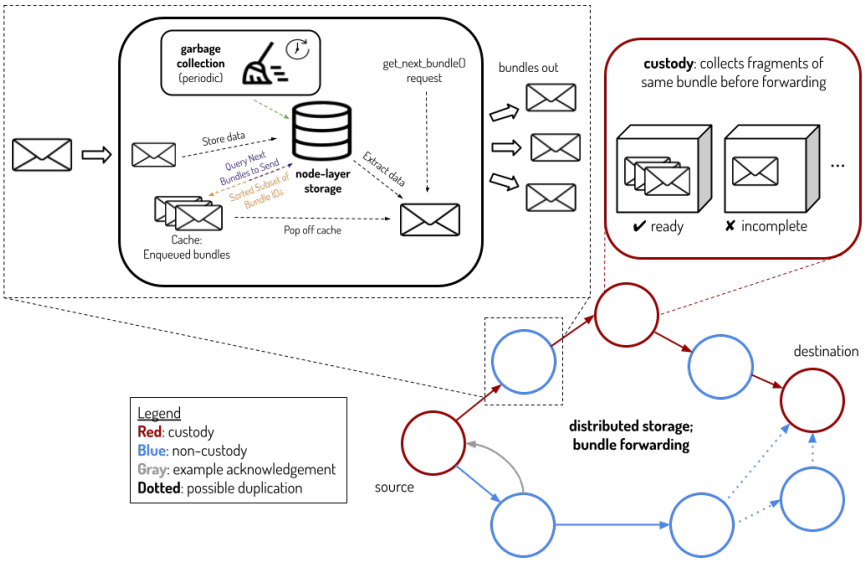

FIGURE 1. System Diagram

1.2. **Assumptions.** We assume that landslide/solar flare emergencies as well as software updates are marked as priority 3, and that the different types of routine communications described in the spec correspond to the 3 different priority levels (emergencies 3, tasks 2, others 1).

1.3. **Added Flags.** We add two new options: `retention_class` and `system_update`. The `retention_class` option will use 2 bits and will be contained in the primary block of each bundle. This bit field signals to the rest of the system the relevancy of the bundle data after expiry. The flag will be assigned by the sending application, where:

- **Most significant bit (MSB):** indicates whether the bundle data remains highly relevant (i.e. deleting it may cause serious issues in the system) within 24 hours of expiry, and
- **Least significant bit (LSB):** indicates the bundle data remains moderately relevant (i.e. very low chance that deleting leads to serious consequences, but it still contains useful data) more than 24 hours after expiry

This new option in the primary flag of bundles will allow us to intelligently garbage collect bundles from storage. In our system, we have two different garbage collection routines, letting the setting of `retention_class` directly indicate the probability that a certain bundle is deleted (refer to 2.2).

Furthermore, the `system_update` flag, also placed in the primary block, will be used to indicate that a bundle is designated for system maintenance. This is primarily because nodes require a confirmation that the updates they have requested have been successfully installed; hence, destination nodes will need to store the `report_to` nodes for system update bundles, in order to send responses once installation is complete (see 2.7.2 for further details).

2. Modules

2.1. **Node-Layer Storage.** Our storage system is designed to efficiently filter bundles based on specific heuristics, enabling fast insertions, updates, and retrieval of bundles that meet defined constraints. It is also designed to store bundle contents both reliably and scalably. To this end, the storage system on each node is comprised of two components: a SQLite database and a UNIX-based file system.

2.1.1. *SQLite Database.* Due to the implicit structure of bundles and the need for efficient filtering and sorting of data based on heuristics, we opt for a persistent in-memory SQLite as it is a relatively lightweight DBMS that meets these performance goals.

The database consists of a single table `bundles` that contains entries for each bundle currently in storage. Each row would contain the following columns:

- `bundle_id` (PRIMARY KEY): integer `Bundle ID` stored in primary block of bundle (4 bytes)
- `priority_level`: integer denoting the priority-level of the bundle (`0 = high`, `1 = medium`, `2 = low`)
- `expiration_date`: datetime object that indicates bundle expiration time
- `retention_class`: integer that denotes the retention class (0-3) of the bundle
- `empty`: an integer column that takes on 0 or 1 (defaults to 0), which is marked by the garbage collector as 1 when the bundle data corresponding to the id `bundle_id` has been deleted, so entry row can be reused for new bundle
- `next_hop`: address of `next_node` returned by `next_hop(destination)`, `null` if no `next_node` is returned

- `fragment_id`: sequence number of fragmented bundle, `null` if not fragmented bundle
- `checksum`: checksum of bundle payload block to be used for data integrity verification
- `size`: size of bundle (in bytes)

2.1.2. *UNIX-based File System.* To optimize the queries on our SQLite database, we avoid storing the entire bundles in each entry for `bundles`. Instead, we establish a one-to-one mapping between each table entry's `bundle_id` and the corresponding bundle data.

To reliably store and manage these bundle mappings, we deploy the UNIX file system (UFS) on each machine on the network. While an alternative like the Zettabyte File System (ZFS) may provide stronger data integrity guarantees and much higher scalability as opposed to a simpler file system like UFS, it is slightly overkill for our needs. ZFS has significant overhead (taking up more than 8 GB of RAM) and unnecessary scalability guarantees (up to 2^{56} entries in a single directory), among many other advanced features (e.g. self-healing data, RAID-Z, de-duplication, etc.).

To bridge the gap between the shortcomings of UFS – in terms of data integrity, reliability, and scalability – and our system requirements, we implement the following enhancements on top of UFS:

- **Disk Mirroring.** For each LEOCom and GEOCom in our system, the storage space is split in half by two slightly different storage devices, which we will denote by `storage_device_1` and `storage_device_2`. At any point in time, over 10% of machines have a storage device down, effectively halving storage for affected machines.

  To remedy this, we mirror the two storage devices, placing an identical UNIX-based file system on each. Whenever data is written to or deleted from a particular storage device, the operation is mirrored on the other. While this halves the available storage for unique bundle data on LEOComs and GEOComs, valuable data is no longer at risk of being permanently lost in the case of an abrupt device failure, which is crucial for system reliability.

  When a device goes down and resets, we initiate a process to repopulate the crashed device storage from the working mirror. This process consists of:

  (1) Reformatting and recreating the UNIX-based file system on the crashed mirror
  (2) Mounting the file systems on each mirror to their respective distinct mount points: `mnt/mirror_A, mnt/mirror_B`
  (3) OS copies the files/directories from the working mirror's mount point to the recovered mirror's mount point

  Furthermore, when reading a file from storage for bundle with id `bundle_id`, we read the data from both mirrors. Using the checksum stored in the table for the row entry with key `bundle_id`, we verify the corresponding checksums. If one mirror matches but the other does not, we overwrite the inaccurate copy to the other mirror's copy. If they are both incorrect, we proceed with the following cases:

  – **Case 1** `system_update = 1`: System update bundles must be preserved and in-tact. A corrupted system update bundle may potentially

introduce severe defects into machine software. Hence, we default to dropping, i.e. deleting, both mirrors' copies of the bundle data from storage.
  - **Case 2** `system_update = 0`: We randomly select one of the mirror's copies and overwrite the other mirror's file to match that copy. For bundles other than system update and maintenance bundles, it is not absolutely necessary for the data to be fully intact or correct. Instead, the timely transmission of "mostly correct" data is prioritized.
- **Filename Hashing.** File system performance for UFS deteriorates as the number of files within a single directory grows large. Hence, we seek to distribute the files evenly under 16 subdirectories, named `./storage/bucket_1, ./storage/bucket_2, ..., ./storage/bucket_16`. To randomly and uniformly distribute bundle files into these buckets, we use a SHA-1 hash. SHA-1 is a lightweight hashing function (can be computed within 1μs), that provides strong enough uniformity guarantees for our use case. For a bundle with id `bundle_id`, we apply a SHA-1 hash to `bundle_id` and apply the mask `0xf` to get the last four bits, which we will denote by `masked_hash`. The corresponding bundle will thus be placed at the location: `./storage/bucket_{masked_hash}/bundle_{bundle_id}.txt`.

2.1.3. *Deletions.* To delete a bundle, the matching `bundle_id` should be provided. We first delete the file for `bundle_id`, i.e. delete the file containing the data for `bundle_id` from both mirrors. Leaving the entry in the database table is okay as each row takes up very little space as we only store 32-bit ids as opposed to the entire block of data that is stored for each bundle in the key-value store, which is deleted. We instead just mark the corresponding entry as `empty = 1` to signal it is free for future incoming bundles.

2.1.4. *Querying bundles.* To fulfill requests to service outgoing queues, we expose the interface:

`query_bundle_ids_from_storage(out_node, queue_priority)`,

where `out_node` provides the address of the neighboring node we are forwarding to, and `queue_priority` specifies the priority level of the particular outgoing queue to `out_node`.

Upon the function call `query_bundle_ids_from_storage(out_node, queue_priority)`, the following SQL query is executed on the `bundles` table:

```
  SELECT bundle_id
    FROM bundles
   WHERE empty=0 AND next_hop=out_node AND priority_level=queue_priority
ORDER BY retention_class ASC, expiration_date ASC, size ASC
   LIMIT 1000.
```

This query filters for bundles that are fit to service the outgoing priority level `queue_priority` queue to `out_node`. By ordering first on `retention_class`, we ensure that bundles that are highly time-sensitive and are least useful post-expiry are given the opportunity to be transmitted before becoming obsolete. A secondary sort by `expiration_date` prioritizes bundles with sooner expiration_dates among bundles with the same `retention_class`. Additionally, we apply a tertiary ordering by ascending size. To improve the throughput and bandwidth utilization

along links, we choose to send smaller bundles first when all else is equal (i.e. `retention_class`, `expiration_date`). We cap the number of IDs to 1000. This is motivated by the fact that each node-node link has a maximum bandwidth of around 1 Gb/s. We assume that the average bundle size is roughly 100 MB for simplicity. Hence, 1000 bundle IDs equate to over 1000×100 MB = 100 GB of data. Given a 1 Gb/s link, it would take just under 15 minutes of continuous transfer at peak bandwidth-link utilization to consume 1000 bundles. At the same time, the list of IDs remains small, 1000×4 bytes = 4 KB, suitable for fast in-memory caching. Additionally, querying a large batch avoids the overhead of having to fetch individual bundles one at a time. We return the resulting ordered list of IDs.

2.2. **Garbage Collection.** Over time, the memory and storage on nodes can accumulate expired bundles, which can take up valuable space on nodes that could otherwise be used for more urgent data. To reduce the bloat from non-critical, expired bundles within the network, we conduct a routine light daily cleanup and a thorough monthly cleanup of memory and storage in the network.

For the daily cleanup, we randomly select a subset (10% - 30%) of all of the nodes to garbage-collect. Doing a daily cleanup on all of the nodes would be very resource-intensive and substantially lower system availability; hence, we only run it on a small portion of machines. Additionally, by randomly sampling, we avoid bias in which nodes are chosen for daily cleanup and are able to ensure fair and uniform coverage across the network. To minimize the maximum load induced on the system as a result of daily cleanup, we stagger the cleanups for each machine.

On the other hand, for the monthly cleanup, we do a thorough garbage collection on each node in the network. To mitigate the stress that this may place on the system, we designate a five-day window each month for the cleanup, in which no daily cleanup tasks are scheduled. As NEBULA monitors traffic throughout the network, it intelligently schedules the cleanups for each machine during dips in network traffic/congestion, making sure to stagger the start times.

Now, using the `retention_class` option (refer to 1.3) we presented earlier, we detail how the garbage collector decides whether or not to delete a bundle:

Table 1. Garbage Collection Diagram

Retention Class	**Probability of Bundle Deletion**	
	Daily Cleanup	**Monthly Cleanup**
`0b00`	100%	100%
`0b01`	75%	75%
`0b10`	50%	100%
`0b11`	25%	75%

This probabilistic deletion approach ensures unbiased removal of bundles of the same `retention_class`. Additionally, by adjusting deletion probabilities for different `retention_class` settings independently for daily and monthly cleanups, we are able to more aggressively target bundles that are less likely to be useful either in a short or long period after their expiry during garbage collection.

Refer to 2.7.1 for specifics on freeing memory in custody nodes.

2.3. **Distributed Storage.** While local storage on each node enables efficient handling of individual bundles, NEBULA is fundamentally a distributed system. As such, it must support coordinated decisions about storage and transmission across nodes. This section outlines our strategy for managing distributed storage to ensure reliable delivery, particularly for high-priority or emergency data. We highlight two key mechanisms: reserved storage for mission-critical bundles and bundle duplication to increase delivery robustness across the network.

TABLE 2. Number of Neighbors Receiving Copy of Low-volume Bundles

Number of Neighbors to Select	**Probability**		
	High Priority	**Medium Priority**	**Low Priority**
0	1/4	1/3	1/2
1	1/4	1/3	1/2
2	1/4	1/3	0
3	1/4	0	0

2.3.1. *Reserved Storage.* For each of the nodes in the network, we reserve a portion of storage to ensure that NEBULA can swiftly and reliably transmit telemetry and routine management data in the case of an emergency. In particular, we designate 50GB of storage on each node as strictly reserved for priority level 3 bundles. We argue that 50GB is enough space to handle land telemetry data (20 files of 2GB), solar telemetry data (20 files of 200MB), and other routine emergency alerts (each of which is only 10MB). Thus, 50GB offers a comfortable buffer to ensure reliability of emergency data.

As a tradeoff, lower priority bundles can only be stored to the node returned by `next_hop(bundle_destination)` or `get_neighbors()` if there are at least 50GB of local storage available in addition to the size of the package. However, they can still be forwarded for those nodes to then decide what to do with.

2.3.2. *Bundle Duplication/Redundant Storage.* If `next_hop(bundle_destination)` does not return any nodes, NEBULA attempts to make copies of the bundle and send a copy to a subset of its neighbors returned by `get_neighbors()`. Using TABLE 2 above, we determine `num_neighbors_to_select` for low volume bundles (routine management data), and we randomly select a subset of neighbors of size

`min(get_neighbors().size, num_neighbors_to_select)`.

For high volume bundles (non-routine management data) that are strictly high-priority, we attempt to send a single copy to the neighboring node with the most available storage (assuming a non-empty set of neighbors) with probability 1/2, as returned by `query_neighbor(size_of_bundle, neighbor)`.

By sending bundle copies to neighboring nodes, we increase the likelihood of a successful delivery to the destination by considering alternative paths in addition to the paths stemming from the original node. Furthermore, by sending only a single copy to high-priority high-volume bundles, we mitigate the overhead associated with constructing the copies, while improving the likelihood of transmission for urgent data. We randomly select the neighbors for lower-volume bundles in order to balance the bandwidth consumption by the transmission of these redundant

packets between each node. On the other hand, high-volume bundles place much more stress on node storage, so we choose the node with the most free storage. To mitigate the possible stress that these high-priority high-volume bundle copies can place on the bandwidth of particular nodes with high storage availability, we only send a copy with probability 1/2 (around half the time).

2.3.3. *Pre-Emptive Deletion.* An additional optimization is pre-emptive deletion, where, if a bundle has been sent to various nodes, we assume it is more likely that the bundle will arrive at its designated node. Thus, this bundle is marked as higher priority for deletion within our sender node through an update to its retention class bit (reference 2.2). The bit update associated with the update is directly related to how many nodes the bundle has been sent to and is detailed by TABLE 3 below. Note that we **only** update the retention class in the event that it decreases the retention class setting; we are looking to reduce the likelihood of retention as much as possible within a reasonable limit to improve the storage availability on nodes.

Alternate Paths	**Retention Class**
0	`0b10`
1	`0b01`
2+	`0b00`

TABLE 3. Retention Class Update for Number of Alternate Paths

2.4. **Bundle Forwarding and Fragmentation.** Bundles may be fragmented (if this is allowed according to bit flag 2) as they are forwarded through the network; this section describes details of the fragmentation and forwarding processes.

Fragmentation is implemented using the `fragment_bundle` function of the Bundle Protocol. To increase reliability, we include a checksum for each fragment created (or each bundle, if it cannot be split). This additional component is required due to the high error rates in the network and desire to minimize retransmissions (details in 2.6.1).

2.5. **Queue Management.** Routing in SolarNet must navigate intermittent links, fluctuating bandwidth, and limited node storage. While the Bundle Protocol provides a baseline store-and-forward mechanism, NEBULA enhances this with more refined queue handling to improve storage efficiency and forwarding speed. Specifically, we take advantage of our freedom to design the internal queue structure and how bundles are accessed once they reach the front of their respective queues. This section outlines NEBULA's approach to queue-to-bundle interfacing and key optimizations that reduce latency and improve throughput under real-world constraints.

2.5.1. *Prefetching bundles.* We maintain an in-memory cache composed of FIFO queues, one for each outgoing link ($\approx$ 300 in total, given 3 per 110 machines). We allocate 5KB of space per queue, so 1.5MB total space which is insignificant compared to the total 64GB of memory. Of this, 4KB is used to store pre-fetched bundle IDs returned by `query_bundle_ids_from_storage` (refer to 2.1.4). The remaining 1KB serves as a reserved region to reinsert IDs for transmitted bundles that failed to receive an acknowledgment within the designated timeout.

When a bundle is successfully acknowledged, it is permanently deleted from storage. Otherwise, if no acknowledgment is received in time, the bundle is not discarded and its ID is instead pushed onto the reserved 1 KB region of the corresponding queue for its transmission to be retried as soon as possible. NEBULA alternates between popping bundle IDs from the 4KB and 1KB regions to prevent the system from starving out the bundles from either section.

We refresh the the 4KB region of each queue every 15 minutes with a new ordered list of bundle IDs returned by `query_bundle_ids_from_storage(...)`. This 15-minute interval was chosen based on the fact that, at full capacity, it takes around $13 - 14$ minutes to consume 1000 bundle IDs (4KB) as described in 2.1.4. So, refreshing every 15 minutes ensures that the queue is almost depleted before fetching a new set of bundle IDs, minimizing the number of calls needed to be made to storage.

When a bundle is requested to service a particular outgoing link, we just pull the next pre-fetched bundle ID from the corresponding in-memory FIFO queue in our cache. We then pull the associated bundle data from storage and enqueue it to the respective outgoing link for forwarding.

2.6. **Bundle Forwarding and Fragmentation.** Bundles may be fragmented (if this is allowed according to bit flag 2) as they are forwarded through the network; this section describes details of the fragmentation and forwarding processes.

Fragmentation is implemented using the `fragment_bundle` function of the Bundle Protocol. To increase reliability, we include a checksum for each fragment created (or each bundle, if it cannot be split). This additional component is required due to the high error rates in the network and desire to minimize retransmissions (details in 2.6.1).

2.6.1. *Fragment Size.* By default, bundles will be initially fragmented at the source node into bundles of size 16384 bytes. Generally, NEBULA maintains fragments no larger than $2^{14} = 16384$ bytes and no smaller than $2^9 = 512$ bytes. This fragment size is chosen to improve reliability, due to the following considerations:

- To reduce overhead and hence increase efficiency, larger fragment sizes are beneficial.
- Since roundtrip times are long, retransmissions (in the event of reported failures/losses) are expensive, so larger fragment sizes are beneficial,
- Node memory limits are relatively large (64 GB), so large fragment sizes are acceptable.
- Error rates are high, so fragments cannot be too large, to prevent losing large chunks of data at a time. However, we also have other methods of ensuring correctness of the transmitted data, e.g. redundancy (see 2.6.2) and checksums.

2.6.2. *Forwarding.* Each node maintains a dynamic routing table that is refreshed at one-minute intervals via the Bundle Protocol. This table supplies next-hop options, guaranteed link capacity, and custody and neighbor information.

For non-custody bundles, the node extracts the destination address from the bundle's primary block and invokes the `next_hop` function to determine the optimal transmission candidate based on current link bandwidth and the estimated communication window. If the remaining sendable capacity for this `next_hop` is more than 10 times the size of our bundle, we assume the traffic is relatively light,

and we transmit a duplicate fragment to this next hop after some delay. This heuristic is chosen (instead of, for instance, a more distributed information-sharing system where each node sends information about its own load to its neighbors) to reduce latency and primarily consider the available capacity of this specific link.

Additionally, we have stored an old `next_hop` value in our node-layer storage table. If this is different from the newly queried `next_hop`, we will try to send our fragment there as well, because there is a chance that it is still reachable but just not the most ideal path currently.

On the other hand, for custody bundles, the transmission procedure diverges. If the current node is not designated as the next custody node (as indicated in the bundle's primary block), the node retrieves the `next_custody_node` field from the primary block and uses the `next_hop` function to forward the bundle toward that custody node (similar to non-custody process).

Conversely, if the current node is identified as the next custody node, it must first assemble all fragments corresponding to the complete bundle. Once reassembly is complete, the node issues a custody acceptance notification to the bundle's originating custody address. It then invokes the `next_custody_node` function using the destination address, updates the bundle's primary block by setting its current custody address to its own address and modifying the `next_custody_node` field accordingly, and resumes transmission (details in 2.7.1).

Upon receiving a custody acceptance signal from the subsequent custody node, the node frees up the related storage. If no acceptance is received within a predefined timeout (based on current routing tables/remaining connectivity times), the bundle is retransmitted, preferably via an alternative custody node, to enhance delivery robustness. More details about the custody nodes can be found in 2.7.1.

2.7. **Custody and Reporting.** NEBULA incorporates two crucial mechanisms to improve reliability of data transmission: custody nodes which guarantee message persistence and facilitate retries, and the reporting mechanism providing feedback on message delivery using administrative bundles.

2.7.1. *Custody Nodes.* Custody nodes in NEBULA adopt 3 key strategies to improve storage and forwarding guarantees:

(1) If a custody node is nearing storage capacity, it prioritizes forwarding custody bundles to less-loaded custody nodes, forming an alternate custody path.
(2) If a custody node is about to lose connectivity to the next hop in the custody path (from routing table), it prioritizes forwarding custody bundles (if calculations allow complete transmission, given bundle size / connection bandwidth / next hop storage capacity) over normal bundles.
(3) Priority 3 custody bundles are replicated across different custody paths (different `next_hop`s and/or `next_custody_node`s), reducing the time for 1 complete copy to reach a new custody node. This applies only to priority 3 custody bundles to balance the speed of reliable transmission with network traffic.

2.7.2. *Reporting Mechanism.* Applications can set reporting flags when calling `create_bundle`, but NEBULA additionally enforces certain bits to be set:

In the context of acknowledgments, there is a key tradeoff between bandwidth usage and communication certainty. Maximum reporting for all bundles would

Bundle Priority	**Receive ack requested (bit 5)**	**Bundle reception status report requested (bit 14)**	**Bundle forwarding status report requested (bit 16)**
1	1	0	0
2	1	1	0
3	1	1	1

TABLE 4. Reporting bit flags for different bundle priorities.

cause significant overhead, but confirmation of important messages greatly improves reliability by informing us if there is a need to resend critical information. Also, different bundles have varying acknowledgment needs: software updates require installation confirmations besides just delivery receipts, while other data types do not explicitly require acknowledgments but do have varying reliability demands.

As such, NEBULA enables reporting bit flags (5 for acknowledgment to direct sender, 14 for reception, 16 for forwarding) according to Table 4, with the report-to node set as the bundle's source. For software updates, an additional report confirms installation completion to the source (report-to) node from destination nodes; we identify software updates using the `system_update` flag proposed in 1.3.

Since reports are consolidated in small administrative bundles, failure chances are low but non-negligible in the unreliable space medium. Resending all administrative bundles a few times incurs far less overhead than resending multiple relevant original bundles (GB or even TB of data) during reporting failure. Hence, NEBULA sends 3 copies of each reporting bundle (possibly along different routes, depending on availability) with slight delays based on expected roundtrip time, increasing the probability of report reception while minimizing unnecessary retransmissions of large bundles.

The design choice to retransmit bundles, even along the same route, involves a trade-off between bandwidth usage and reliability. A single transmission may fail due to a wide variety of reasons; while a number of these affect the entire link as a whole, some may only affect the specific bundle or a part of the link at a certain time, such as transient link disruptions due to variations in the environment like radiation or solar activity. Hence, there is still value in retransmitting even along the same link after some delay. This is especially true for small administrative bundles, which use very little bandwidth per bundle, but also applies to other types of bundles (e.g. 2.6.2).

Note that the reports are grouped by their destinations (report-to node or previous hop of the original bundle, depending on type of report). If an administrative bundle is destined for a node that is currently directly reachable (in terms of position and available bandwidth), at least one of the 3 copies will be sent via the direct path, and the control-plane queue will be used in attempting to transmit this administrative bundle directly.

We recommend that the control-plane queue be handled with top priority, that is any bundles currently in the control-plane queue should be handled before bundles in any of the data-plane queues. This is because control-plane bundles should be small and infrequent, so doing this should not affect the data-plane bundles too greatly. Meanwhile, it helps determine bundle transmission statuses sooner (we get reports of successes earlier and can reduce the timeout before retransmission of

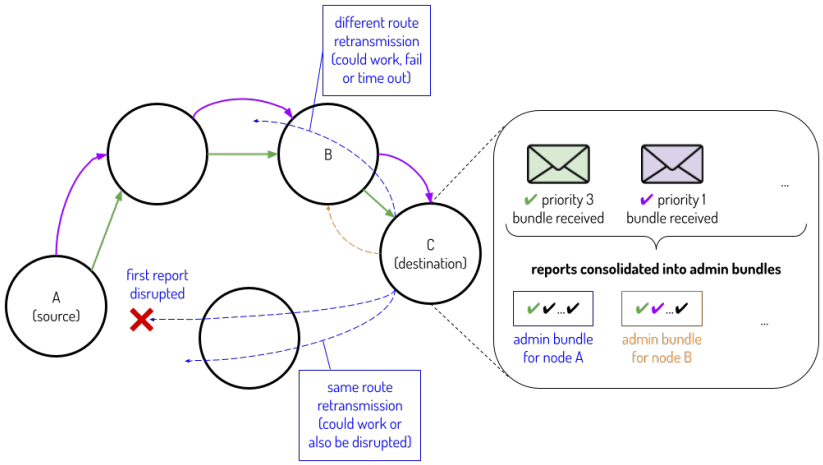

FIGURE 2. Example: transmission and retransmission of administrative bundles for reporting

failed bundles), so bundles can be retransmitted sooner if required, contributing to timeliness of successful bundles reaching their final destinations.

3. Use Cases

3.1. **Software Updates.** Ensuring timely, complete software updates at all nodes is crucial for system integrity and performance. Software updates are assumed to have the highest Bundle Protocol priority level since it is critical that installation begins within 30 minutes, so they will be handled with maximum reporting and sent along the highest priority queues, ensuring successful transmission and acknowledgments. NEBULA also enhances this using the `system_update` flag (1.3). This allows retransmissions to happen quickly if necessary acknowledgments are not received, minimizing update failures that may disrupt the whole network.

3.2. **Land and Solar Telemetry.** Landslides in remote areas like Kyrgyzstan pose significant threats to human lives, while solar flares can severely impact communication systems. High-priority land/solar telemetry data that indicates potential emergencies is handled with maximum reporting and sent along the highest priority queues in NEBULA, which improves speed and reliability of their dissemination; this enables timely, accurate predictions and warnings. Meanwhile, regular telemetry data is sent with lower levels of priority and will not arrive as quickly as emergency data.

For emergency land telemetry data, evidence of landslides can be seen up to 10 days before (hence, most likely on the order of a few days before disaster) and needs to reach the ground 20 minutes before disaster strikes (10 minutes to analyze data and 10 minutes for population to evacuate). The set of 20 2GB files will be fragmented as in 2.6.1 if allowed (≈ 2.6 million 2^{14}-byte fragments) and transmitted to Earth from the land telemetry LEOs, likely with duplicates along different paths as described in 2.3.2.

For emergency solar telemetry data, solar loops may be detected a few hours before a solar storm emergency; a set of 20 consecutive 200MB files will be fragmented as in 2.6.1 if allowed ($\approx$ 256000 2^{14}-byte fragments) and transmitted from the solar telemetry GEOs to Earth. Notifications may also be sent to space crews on Mars and the Moon.

3.3. **Routine Communication.** Emergency alerts in routine communications (from Earth/satellites to space crews) are handled similarly to high-priority telemetry data to avoid severe disruptions or impacts to space crew safety, which directly affect system functionality/performance. Other regular communications (between space crews and Earth) are supported but may experience longer transmission times or more retries compared to critical messages. We expect routine communications to consume less bandwidth overall compared to the other categories of data, due to smaller file sizes and generally lower traffic.

3.4. **Affected Groups.** The main groups of people we had in mind when designing our system were the space crews, ground crews and scientists as well as people who might be affected by the data such as citizens of Kyrgyzstan. We primarily wish to prioritize safety of space crews (i.e. astronauts on the Moon/Mars) and ensure they can stay connected to (or at least informed of disruptions in) the rest of the network; we do so by maintaining availability and reliability of the network, aiming to get critical software updates completed in time and prioritizing emergency solar flare notifications. Many important operations and functions of the network depend on these crews and the facilities they operate, and it is difficult and time-consuming to deploy new people to these roles.

The citizens of Kyrgyzstan are also prioritized since landslide emergency data is given the highest priority level. However, the scientists using telemetry data are generally deprioritized as non-emergency data may take longer to reach ground stations on Earth than the emergency data.

4. Evaluation

4.1. **Quantitative Metrics.** On any given transmission between two nodes, there is an expected 1% chance that this transmission fails. Consider our administrative bundles, which are sent 3 times. Given this expected value, the likelihood that all 3 administrative bundles fail is 0.0001%.

Let us consider how our pre-emptive deletion strategy affects system guarantees. If there are 0 alternate paths, a bundle is marked with retention class $0b10$, which corresponds to a 50% probability that the bundle will be deleted within the day. In the 1% that the bundle transmission fails, there is at least a 50% chance that the bundle is resent again, giving that:

$$P(\text{Bundle Transmission Fails}) = P(\text{Transmission\_1 Fails}) * P(\text{Transmission\_2 Fails})$$

$$= P(\text{Transmission\_1 Fails}) * [P(\text{Bundle Deleted}) + P(\text{Bundle Not Deleted}) * P(\text{Fails})]$$

$$\leq 1\% * [50\% + (50\% * 1\%)] = 0.505\%$$

Essentially halving the likelihood that the transmission fails. When a bundle is sent along one alternate path, it is marked with retention class $0b01$, corresponding to a 75% guarantee that the bundle will be deleted within the day. Therefore,

$$P(\text{All Bundle Trans. F}) = P(\text{Path\_1 Fails}) * P(\text{Path\_2 Fails})$$

$$\leq (1\% * [75\% + (25\% * 1\%)]) * (1\% * [75\% + (25\% * 1\%)] =$$
$$(0.7525\%)^2 \leq 0.005663\%$$

When a bundle is sent along at least 2 alternate paths, it is marked with retention class $0b00$, corresponding to a 100% guarantee that the bundle will be deleted within the day. Therefore,

$$P(\text{All Bundle Trans. F}) = P(\text{Path\_1 Fails}) * P(\text{Path\_2 Fails}) * P(\text{Path\_3 Fails})$$
$$\leq 1\% * 1\% * 1\% = 0.0001\%$$

The same value as for our administrative bundle. Clearly, our pre-emptive deletion does not negatively affect our best-effort design, allowing our system to avoid storage overhead while also offering increasingly impressive guarantees as the number of alternate paths a bundle is sent on increases.

4.1.1. *Worst-Case Scenarios.* Assuming the connection between Earth and Mars is disrupted due to repositioning of the relay, communications from Earth to Mars would need to travel from Earth to a LEOCom to a GEOCom and then to a Mars relay before reaching Mars. The frequency at which this occurs depends on the revolution times of Earth/Mars and this would take longer than the worst-case 20 minutes between Earth and Mars. The largest set of data that would need to be transmitted is the 10 10GB files for Moon/Mars equipment software updates once a month; this needs to happen within 30 minutes. Assuming full custody at each of the intermediate nodes, these files would need to travel along the 2GB/s connection between Earth and LEOCom, 1.2GB/s connection between LEOCom and GEOCom, 622MB/s connection between GEOCom and relay and 622MB/s connection between relay and Mars. If the software update is the only bundle of priority 3 currently, this would take $(100/2) + (100/1.2) + (100 * 2^{10}/622) + (100 * 2^{10}/622) \approx 463$ seconds or approximately 8 minutes (accounting for bundle header overheads). See 4.2 for a comparison of NEBULA's redundancy versus if we had to keep waiting for acknowledgments to time out (the latter could easily double the overall transmission time between consecutive nodes if the data or acknowledgment bundle is lost).

Next, we examine the performance of locating a piece of data on local storage. As we are pre-fetching bundle IDs and placing them onto an in-memory cache, pulling a piece of data from local storage simply entails popping a pre-fetched bundle ID from a queue in our cache and locating the corresponding file from our file system with the bundle contents. As file reads are significantly more expensive than reads to our in-memory cache, the latency associated with locating a piece of data on local storage is defined by the time it takes for a file read. Typical read speeds for UFS is around 200 MB/s to 500 MB/s for a solid-state drive (ssd). Under the assumption that bundles are on average around 100MB (less conservative estimate), it would take from 0.2 - 0.5 seconds to query a bundle from local storage on average. In the worst case, files can be as large as 16GB as with system maintenance/update bundles for GEOComs. It would take between 30 and 80 seconds in order to fully read the bundle from storage. However, these updates only occur twice a month so this does not pose a serious concern for overall system latency/throughput.

As for pre-fetching bundle IDs, we conduct a brief analysis on the latency of the `query_bundle_ids_from_storage(...)` function call. The performance of the function is defined by the speed of execution of the SQL query. As we are using an in-memory persistent database, apply indexes on the columns that we filter

by, have around 30 bytes per row, and have theoretically up to 2^{32} rows, we can expect the query to execute in anywhere from the sub-second to few seconds range in the worst case, similar to the file read. We execute this call once every 15 minutes to avoid having to query an individual bundle ID each time we service an outgoing queue, greatly improving the overall throughput and latency associated with querying bundles from storage.

4.2. **Consideration of Design Choices.** The communication overhead of NEBULA is quite significant, as it includes retransmission of some data bundles and small administrative reporting bundles, adding complexity and increasing bandwidth usage.

This extra communication has been justified throughout our paper as it enhances reliability and ensures that important data, such as software updates or emergency data, can be correctly and fully delivered on time, due to lower probabilities of the first round of transmissions failing (when data bundles are duplicated) and shorter times needed to determine if retransmission is necessary (when reporting bundles are duplicated). Without these retransmissions, we would greatly increase the chances of timeouts being reached and having to trigger a data retransmission before an acknowledgment is received (chances of data and reporting transmissions both failing increase). This means data would take longer to fully travel through the network to its destination. In contrast, in NEBULA, if one of the initial duplicate transmissions was successfully received by an intermediate node, and transmission can immediately continue from there instead of waiting for the timeout and retransmitting from the previous node.

We also mitigate against the negative effects of the system's reliance on duplicate transmissions/storage by restricting replication mainly to certain high-priority bundles or when the bundle is relatively small compared to the available capacity.

4.2.1. *Limitations of NEBULA.* However, the redundancy of both data and reporting bundles in NEBULA is indeed its main limiting factor, as the system will likely perform poorly with higher volumes of data and/or more consistently high traffic, especially if this traffic is high-priority.

In particular, an increase in number of users may cause an increase in volume and regularity of routine communications; if this significantly increases the percentage of traffic from routine communications, the system may be overwhelmed due to the duplication of such bundles (which we assumed to be low-volume) in 2.3.2. This could be mitigated by no longer duplicating routine management data to ensure there is enough bandwidth in the system for more (other types of) data to be transmitted (if the data cannot even be transmitted, there is no point in discussing reliability/timeliness of the transmission).

An increase in software updates specifically could also limit NEBULA, due to the considerable increase in reporting overhead. The effects of this are likely not as severe as an increase in number of users, since reporting bundles are small. However, more software updates may also mean more downtime of individual nodes during updates.

5. Conclusion

NEBULA provides a robust extension to the Bundle Protocol, addressing SolarNet's primary challenges: constrained storage, long transmission delays, and

intermittent connectivity. Through intelligent routing, distributed storage, and a comprehensive reliability framework, NEBULA ensures efficient data transmission throughout the network.

Our design leverages weighted round-robin scheduling, deadline-based prioritization, and a dedicated `system_update` flag to optimize the handling of critical data, such as software updates, emergency telemetry, and mission-critical messages. Custody and reporting mechanisms ensure successful delivery and acknowledgments at key nodes.

5.1. **Future Work.** While NEBULA significantly improves SolarNet's performance, possible improvements or future work could explore:

- Enhancing reporting beyond priority levels, incorporating data type or other features.
- Refining timeout strategies for retransmissions of data/administrative bundles: more detailed and specified methods could improve delivery times, but we did not have any specific strategies in mind and were not sure if such decisions would be significant.
- Investigating whether routing tables can provide all the information desired in 2.7.1, or if alternative methods are needed to obtain such insights.

By addressing these areas, future iterations of NEBULA may further enhance extraterrestrial communication reliability and efficiency, ensuring robust and scalable support for deep-space missions.

6. Author Contributions

All team members worked together to understand the requirements, design the system and review the paper before submission. Diagrams were made by Jamie and Jorge. The first 3 modules were mainly designed by Noah and Jorge, the 4th module was designed by all of us and the last 2 modules were designed by Jamie. In terms of report-writing, generally the first 2 modules (node-layer storage, garbage collection) were written by Noah and the next 2 modules (distributed storage, queue management) were written by Jorge, while the last 2 modules (bundle forwarding and fragmentation, custody and reporting) were written by Jamie. Some of the content in bundle forwarding and fragmentation were inspired by work done by our ex-member Shafick. Noah and Jorge primarily worked on quantitative evaluations while Jamie worked on qualitative evaluations.

7. Acknowledgments and References

We would like to acknowledge Katrina LaCurts, our lecturer and recitation instructor, as well as Jessie Stickgold-Sarah, our WRAP instructor, for their time, feedback and guidance which helped us to better understand improve our system and report. We would also like to thank our TA Umang Bansal for her support and the course staff who diligently helped answer questions on Piazza.