A Distributed Data Store in Orbit

Ott, Jörg; Kangarharju, Jussi; Mohan, Nitinder

DOI
10.1145/3748749.3749093

Licence
Unspecified

Publication date
2025

Document Version
Final published version

Published in
LEO-NET 2025 - Proceedings of the 2025 3rd Workshop on LEO Networking and Communication, Part of SIGCOMM 2025

Citation (APA)
Ott, J., Kangarharju, J., & Mohan, N. (2025). A Distributed Data Store in Orbit. In *LEO-NET 2025 - Proceedings of the 2025 3rd Workshop on LEO Networking and Communication, Part of SIGCOMM 2025* (pp. 64-73). (LEO-NET 2025 - Proceedings of the 2025 3rd Workshop on LEO Networking and Communication, Part of SIGCOMM 2025). Association for Computing Machinery (ACM). https://doi.org/10.1145/3748749.3749093

A Distributed Data Store in Orbit

Jörg Ott
Technical University of Munich
Germany

Jussi Kangarharju
University of Helsinki
Finland

Nitinder Mohan
Delft University of Technology
Netherlands

ABSTRACT

Emerging Low Earth Orbit (LEO) satellite constellations have been considered for uses beyond plain Internet access, including content caching and edge computing. Assuming satellites are equipped with inter-satellite links, we propose using these links and thus the space in-between satellites, paired with a dedicated satellite queuing system, to "store" data and provide access by keeping data in constant flux around the globe. We describe the properties and explore the capabilities of such a system and discuss some potential uses.

CCS CONCEPTS

• **Networks** → **Network architectures**; **Network protocol design**; **Network simulations**.

KEYWORDS

Low Earth Orbit Satellite Systems; Distributed Storage

ACM Reference Format:
Jörg Ott, Jussi Kangarharju, and Nitinder Mohan. 2025. A Distributed Data Store in Orbit. In *3rd Workshop on LEO Networking and Communication (LEO-Net '25), September 8–11, 2025, Coimbra, Portugal.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3748749.3749093

1 INTRODUCTION

As Low-Earth Orbit (LEO) satellite megaconstellations extend the reach of the Internet, speculations about further use beyond plain Internet access arise. These include ideas about caching content in space [7, 13], CDNs [4], computation in orbit [2, 11], edge computing in space [5, 8], and application-specific data aggregation, e.g., in the context of scientific data sensing from space [5]. Building satellite-based content delivery systems would usually require some control plane to locate content objects, route requests, perform load balancing, etc. and possibly actively manage content caches [4].

In this paper, we take a different route: we continuously move content objects, represented as a series of packets, around in the orbital shell, i.e., across all satellites on all orbits at the same altitude of a megaconstellation, one orbit at a time, so that satellites can just wait for objects, eliminating the need for active discovery. We leverage the large distances in space and the high data rate of inter-satellite laser links, which together yield a substantial bandwidth-delay-product, to "store" content as *data in flight* between satellites in addition to providing extra storage on each satellite. In a sample configuration (see fig. 3 and 4), a single content object would traverse all 1,584 satellites (22 in each of 72 orbits) of the lowest Starlink orbital shell at 550 km altitude approximately every 11 seconds, covering a total distance of 3.27M kilometers.

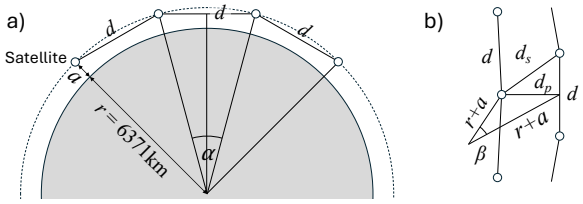

Figure 1: a) Distance d between two satellite in the same orbit and b) d_s between the two closest satellites in adjacent orbital planes.

We present our assumptions and system model in §2 along with the resulting basic system properties, especially the tradeoffs between storage capacity, content replication, and access latency. We discuss mechanisms for adding, replicating, deleting, and routing content objects in §3 and show via simulations how far simple local algorithms with limited state and a lightweight protocol can carry in §4. We conclude with a sample usages and a discussion of possible directions in §5.

2 DESIGN

We assume a LEO satellite constellation in a single orbital shell that covers the inhabited Earth surface with sufficient satellite density that each point on the ground within the coverage area is served by at least one satellite at any given instant. Each LEO satellite is equipped with four lasers to establish ISLs at data rates of $R_{ISL} = 100$ Gbps. Two of the ISLs connect each satellite to its preceding and succeeding neighbors in the same orbital plane, the remaining two connect to satellites in adjacent orbits, creating a `+grid` topology [3]. We assume that the satellites have sufficient power to operate their ISLs continuously, which appears reasonable as current LEO deployments use ISLs. We only consider the operation of a single orbital shell but our considerations can be extended to multiple shells.

Each user terminal connects to exactly one satellite at a time. The terminal reaches the Internet via a bent pipe to a ground station and point of presence (PoP) of the satellite operator. We assume that Internet traffic is routed to the closest ground station and does not traverse many ISLs, minimizing ISL use by end user traffic, so that the satellite operator may bound end user ISL traffic and dedicate an ISL capacity share to *storage traffic* f_s.

Satellites of an orbital shell O are at an altitude a and at an inclination γ_o. The shell is comprised of N_p orbital planes with N_o satellites per orbital plane (or: orbit), assumed to be evenly spaced at an angular distance of $\alpha = 360°/N_o$. With a mean Earth radius of $r = 6371$ km, this yields a distance (line of sight) of $d = 2(r+a)sin(\frac{1}{2}\alpha)$ as shown in figure 1a. The distance between satellites

LEO-Net '25, September 8–11, 2025, Coimbra, Portugal

ACM ISBN 979-8-4007-2090-1/25/09
https://doi.org/10.1145/3748749.3749093

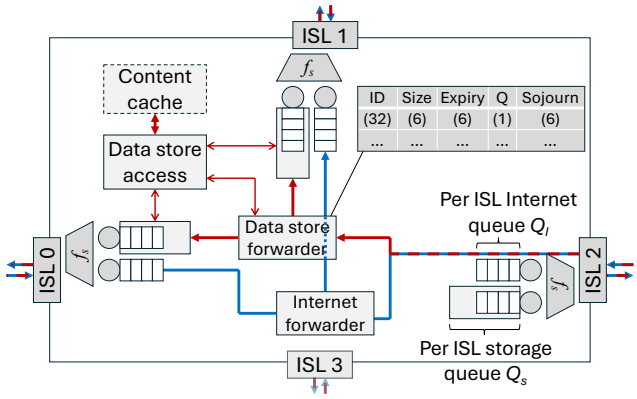

Figure 2: Satellite architecture showing three ISLs with their outbound queues for Internet and storage traffic and the respective forwarding logic, content object state table, and an incoming flow from ISL 2. A data access module matches incoming requests (not shown) against stored content objects with an optional cache.

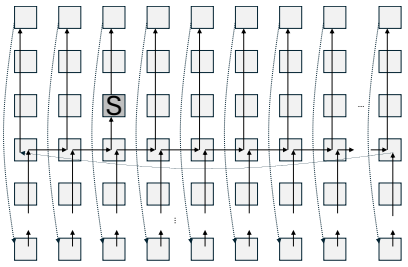

Figure 3: Simple propagation pattern of a single data object from its originator (S) first along each orbital plane and then in a defined area to the next plane.

on adjacent orbital planes varies, being largest at the equator. Figure 1b illustrates the maximum distance d_s between the closest satellites on neighboring planes: the planes are spaced an equal angles of $\beta = 360°/N_p$, yielding a distance of $d_p = 2(r+a)sin(\frac{1}{2}\beta)$ (at the equator), and the satellites are at a maximum phase shift relative to each other, i.e., at an offset of $\frac{d}{2}$ within their respective orbits. The resulting maximum distance at the equator is then $d_s \approx \sqrt{\frac{1}{4}d^2 + d_p}$. To reduce the system dynamics to be accounted for and to obtain a conservative estimate, we assume the maximum distance d_s for computing the propation delay and we ignore that storage capacity of the links across planes.

Consider the Starlink orbital shell at $a = 550$ km altitude with $N_p = 72$ orbital planes at an inclination of $\gamma_o = 53°$ and $N_o = 22$ satellites per orbit. The satellites on an orbital plane are spaced $d = 1969.72$ km apart, at the equator, the distance between two adjacent orbital planes is $d_p = 603.78$ km, and the max distance between the closest satellites on neighboring orbital planes is $d_s = 1155.29$ km.

2.1 System model

Figure 2 sketches part of a single satellite system focusing on the ISLs, i.e., not including the radio links to the ground stations and terminals. Each satellite, at a minimum, provides two send queues of different sizes per interface: one for regular (= Internet) traffic of size Q_I and one for storage traffic of size Q_S. The queues receive proportional treatment when scheduling packets for transmission on an ISL with a share f_s reserved for storage traffic, e.g., using weighted fair queuing.

A content object maintained in the data store comprises a sequence of packets, each packet including at least the content id (e.g., a hash) and total size, its data offset into the object, its expiration time, the total number of replicas and the replica instance number, and a (cryptographic) checksum. A meta data table on the satellite records per content object the id (32 bytes), size (6 bytes), expiration time (6 bytes), sojourn time (6 bytes), and the storage queue (Q, 1 byte), plus internal maintence data, yielding some 64 bytes per record, so that 1M entries would only consume ≈60 MB of memory.

The figure also shows packets incoming from ISL 2 that are classified into Internet and data store traffic and then handled by independent forwarding units that determine their respective next hops, e.g., ISL 0 or 1: the Internet forwarder implements regular L2 switching or IP routing while the data store forwarding logic comprises algorithms to ensure that content object packets circle through all satellites of an orbital shell, to insert, possibly replicate, and delete packets, and to perform error control. We will return such algorithm in §3 and assume for now that a simple one exists that forwards each packet along one orbit at a time and then shifts to the next plane, as illustrated in figure 3.

The data store access unit interfaces to all storage queues and responds to content object requests from satellite terminals (not shown). The data store forwarder records which content objects are in which ISL storage queue along with its expected sojourn time. Given sufficient capacity, a record could be kept for all objects in orbit until they expire and hold the # replicas per object (cf. §2.2) and when it last passed through this satellite. This tells the data store access unit from which queue to fetch the content object, whether it would be feasible to retrieve it from a nearby satellite, or to predict when when its next pass through this satellite is expected.

Content requests would, by default, arrive via the radio interfaces to the ground, but they could also be forwarded by neighbors if those cannot satisfy a request immediately or in the near future. We leave this cooperative caching-style optimization using known techniques for future work.

2.2 Basic properties and trade-offs

We now take a brief look at the theoretical properties of our design. With content objects in constant flux around the globe, two performance metrics are of particular interest: the total *storage capacity*, C_o, of the orbital shell and the *content periodicity*, t_a. The periodicity t_a is defined as the time elapsing between a given content object (or one of its replicas) passing twice through the same satellite and serves as a rough approximation of the worst case access latency from a terminal connected to that satellite.

For storage capacity, we consider: 1) the storage queue for each ISL transmitter, Q_S, and 2) the data in flight on each ISL, $F_s = \frac{d}{c} \times f_s \times R_{ISL}$, as per the link's bandwidth-delay product for the

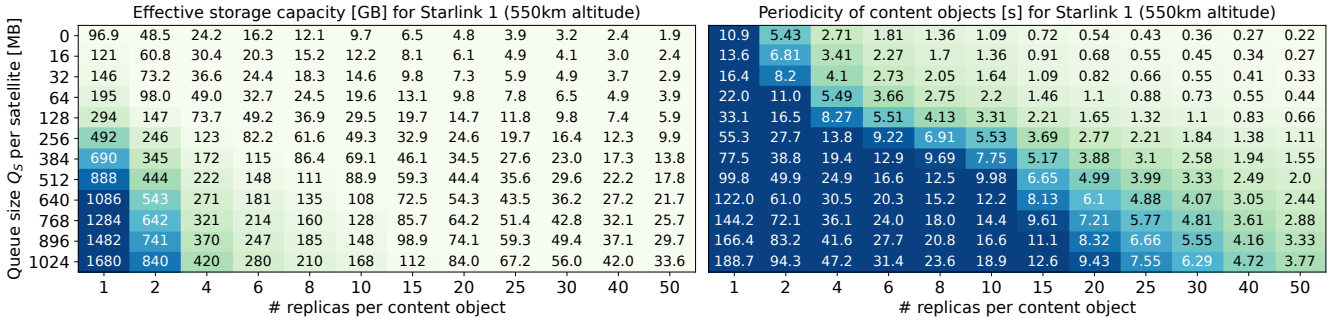

Figure 4: Effective storage space and periodicity of a content object passing through the same satellite as a function of the number of replicas and the content queue size per satellite for Starlink orbital shell at 550km altitude.

traffic share f_s, with $c = 299,792.458$ km/s. This yields a capacity per satellite (including one outgoing ISL within its orbit) of $C_s = Q_S + F_s$, per orbital plane of $C_p = C_s \times N_o$, and per orbital shell of $C_o = C_p \times N_p$. Due to their potential variability, we do not account for the capacity added by links across orbital planes. If K replicas of content objects are kept, the effective storage capacity of the shell is $C_e = \frac{C_o}{K}$.

For the latter, access latency, we compute how long it takes a single content item to pass through the entire orbital shell, t_o. We consider the processing and queuing delays per satellite, t_f and t_q, respectively, as well as the propagation delay for each link within an orbital plane, t_p, and across orbital planes t_s. We assume $t_f = 0.1$ ms for each content object, which roughly matches the interarrival time of 1 MB sized objects at $R_{ISL} \times f_s = 80$ Gbps and allows for sufficient local processing and state management. We obtain the maximum $t_q = \frac{f_s \times Q_S}{R_{ISL}}$ and the propagation delays as $t_p = \frac{d}{c}$ and the worst case $t_s = \frac{d_s}{c}$. This yields the maximum time for a content object to pass through all satellites of a shell, the *rotation time*, t_o, as

$$t_o = N_o \times (t_f + t_q + t_p) + N_p \times (t_f + t_q + t_s)$$

for the above simple propagation pattern (fig. 3). As for storage capacity, we may assume K replicas of a content item evently spaced across all satellites, which would reduce the periodicity, i.e., the effective access latency to $t_a = \frac{t_o}{K}$.

Figure 4 shows the effective storage capacity C_o (left) and effective periodicity t_a (≈ access latency, right) for the Starlink orbital shell 1 at 550 km altitude. Assuming a target access latency of <10 s, we see that up to 20 replicas would be needed, whereas staying below 30 s 1–8 replicas suffice.

There is an obvious tradeoff between capacity and periodicity. Combining both, we may set a target periodicity and derive the necessary number of replicas for a given storage queue size, from which we can then compute the effective storage capacity. Doing this for all megaconstellations as per table 2 in Appendix B, we show the effective capacity of the data store for a target periodicity of 10 s as a function of the queue size in figure 5 (top). We find that increasing the storage queue size yields an increasing capacity, albeit not monotonically. The capacity growth is expectedly more pronounced for shells with fewer orbits (Starlink 3, 4, 5 use just 8, 5, and 6 orbits, respectively) as less data is "in flight" between satellites on fewer orbits. Fluctuation appears stronger for those

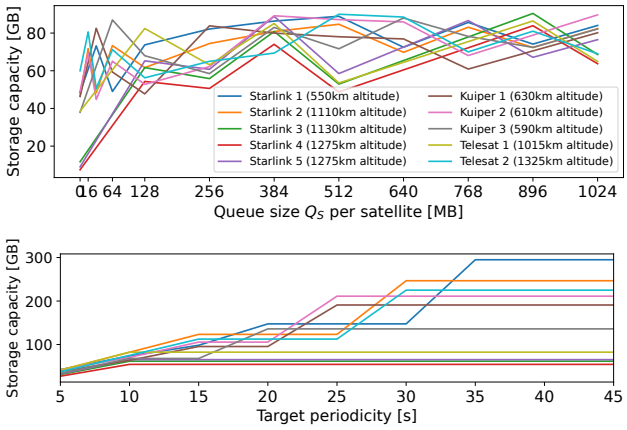

Figure 5: Storage capacity for the orbital shells of different megaconstellations for a target periodicity of 10 s (top) and for varying periodicity at 128 MB queue size (bottom)

shells with fewer satellites. As increasing queue capacity leads to longer object rotation times t_o, this needs to be compensated by additional replicas so that we observe dimishing (if any) returns. The capacity grows roughly linearly as a step function with increasing periodicity (figure 5, bottom). Overall, we obtain some 40–80 GB storage capacity for a periodicity of 10 s pretty much independent of the megaconstellation.

Above, we assume sending the content objects into one direction through the orbital plane, hence only considering half the available transmission and storage capacity. Using both directions would thus duplicate the available capacity and further reduce the latency; however, the intervals at which content objects pass through a satellite would no longer be uniformly distributed; this is left for future study.

3 OPERATION

So far for some theory. We now introduce a strawman algorithm to realize a baseline for storage in space, with routing (§3.1), inserting/deleting content (§3.2), and error handling (§3.3). We assume a system without malicious nodes which appears reasonable for closed satellite systems as today's megaconstellations are.

3.1 Steady state operation

A simple strawman routing algorithm as alluded to above makes a content object traverse one orbital plane and then, upon reaching the satellite it started at, move to the next orbital plane, shown in figure 6. We number the ISL interfaces clockwise as in figure 2; the odd numbered ones are for the North-South directions to satellites within the same orbit, the even numbered ones for East-Wast across orbits.

We assume that each storage packet carries two "hop" counters: n to indicate the count of the satellite in the present orbit and p to count the orbital plane: n allows determining when all satellites of a given orbital plane were visited, p determines when the entire orbital shell is covered. n is incremented whenever a packet is received from a satellite within the same orbital plane. If n reaches N_o, n is reset to 0 and the packet is passed on to the next plane, incrementing p; otherwise, the packet is passed on to the next satellite within the same plane (cf. fig. 3). This works in either North-South direction.

```
Input:  Packet pkt, Incoming interface # if_in
Return: Outgoing interface #

incomding_storage_pkt (pkt, if_in) {
    if (if_in %2 == 1) {
       // packets coming from a different orbital plane
        if (++pkt.p == N_p)
            pkt.p = 0;
        return (if_in + 1) % 4;
    }
    // packets coming from within the orbital plane
    if ((++pkt.n) == N_o) {
        pkt.n = 0;
        return (if_in + 3) % 4;
    else
        return (if_in + 2) % 4;
}
```

Figure 6: Simple routing algorithm to make a content object pass through all satellites of an orbital shell.

Over time, the algorithm needs to adjust n to ensure that a packet is always passed to a neighboring orbital plane when the sending satellite is close to the equator. This is easily achieved as the satellite can observe its own position and update n by ± 1 to pre- or postpone switching orbital planes.

Obviously, other (smarter) algorithms are conceivable within the limitations of the time-varying network topology. The task that each content item should traverse each satellite within the orbital shell translates into the well-known traveling salesman problem applied to a grid-like topology with recurring visits. Interesting questions could be how to use multiple salespersons (i.e., object replicas) to optimize the distribution of the time between repeated visits by the same or different salespersons for a given (non-)uniform utility function.

3.2 Adding and removing content

To add a content object into the distributed data store, the originating satellite defines the object metadata: id I, expiration time t_{ttl}, and replication factor R. The id I could be a cryptographic identifier of the creator plus a hash to identify the content, e.g., like a CID in IPFS [1]. Transmission of the replicas is evenly spaced by observing the current rotation time t_o of a single object , i.e., the time it takes the object to pass through all satellites of the orbital shell, and dividing this by R; each replica is associated with a unique instance identifier relative to I, e.g., by adding a replica count I_R, $0 \leq I_R < R$. The pair (I, I_R) can then be used to sample t_r. To send the content object, it is split across packets, each of which carries the metadata as a header (plus the offset for the packet) and enqueued into the storage queue(s) of its local ISL 0 (and ISL 2) interface(s).

The originating satellite adjusts the initial value chosen for n based on its own position to ensure that the object is forwarded to the next orbital plane close to the equator: setting $n > 0$ reduces the number of hops before switching orbital planes; in this case, the object will not do a full circle through all of the initial orbit. The creator may retain a copy of the object after sending R replicas for later repair.

To determine if there is room for further insertions, satellites observe the object rotation time t_o that represents the aggregate fill level of the overall storage system.[1]

An object is deleted when its lifetime t_{ttl} expires, for which we may assume coarsely synchronized clocks across all satellites The originating satellite may also explicitly delete a content object: it simply stops forwarding any packets belonging to this object. With our strawman routing algorithm, this would deterministically clear the object from the orbital shell within t_o as all packets pass through all satellites. Other forwarding algorithms may have to rely on the expiry time or craft explicit deletion packets ("anti-packets" [12]).

3.3 Error handling

Communication is subject to errors that lead to packet losses. Such errors include 1) bit errors on the laser link, 2) tracking/pointing errors when tracking the neighboring satellites, and 3) the reconfiguration time when connecting to a different neighboring satellite. While 1) may lead to individual packet losses, 2) could incur short loss bursts and 3) extended link unavailability and hence longer loss bursts or delays.

1) could be overcome by applying FEC (e.g., simple XOR, Reed-Solomon codes) mechanisms and ensure that all packets of an object can be recovered at each hop to avoid error accumulation. If content objects are small relative to the storage queue size, recovery could happen while the object is queued and thus not negatively affect periodicity. For 3), a simple approach is predicting when a reconfiguration of a satellite link is to occur and route to the next orbital plane one satellite earlier or later. The rerouting satellite needs to consider its own and its neighbors' queue capacity to not affect the overall stability of the system.

Concerning 2, assuming the above FEC mechanisms won't suffice to cope with an outage, a simple fallback mechanism could be used: if restoring a replica fails, the satellite in question could just drop all its packets and rely on the source node to re-instantiate the replica. For this, the creator monitors if still all R replicas of its objects circulate and recreates the missing ones if a gap in unique replica numbers is detected. This would just cause a temporary spike in periodicity and repair would happen within $O(t_o)$. Originators could also send content objects using rateless codes (e.g., fountain codes) for initial redundancy and add further encoding symbols in response to observed losses.

[1] This somewhat coarsely resembles FDDI token ring networks that use a *target token rotation time* to determine the current utilization of the ring and thus if an attached station is allowed to send data [9].

4 INITIAL EVALUATION

To understand how does the above strawman design works "in practice", we developed a custom simulator in Python that implements the connectivity pattern of fig. 3 and the forwarding algorithm of fig. 6. Objects are forwarded as messages and only processed by the receiver after they were received in full, for which a dedicated receive buffer is available. Once received, an object is moved to the next hop storage queue provided that there is enough space; otherwise it waits in the receive buffer. Objects sourced at a satellite are kept in a source queue and wait there until there is enough space in the next hop queue. Forwarding objects takes precedence over introducing new locally sourced ones. In our simulations, we introduce dedicated tokens (size: 1024 B), one per orbital plane, to continuously measure the rotation time. For our initial evaluation of the basic system properties, we consider only a period with stable satellite topology. Simulations operate in ticks of 1 ms and collect statistics every 10 ms.

We are interested in how the system stores incoming objects and how the rotation time evolves compared to our theoretical considerations. Therefore, we simulate the ramp-up phase from an initially empty system until its theoretical storage capacity is exhausted. We use a warmup time until all tokens have rotated once through the constellation so that each satellite can independently compute the rotation time. After the warmup, we start generating objects of fixed or variable size in fixed intervals (just one replica per object, $t_{ttl} = \infty$) at randomly chosen satellites (uniform distribution). We observe the data volume of the objects that are effectively stored in the system, the evolution of the rotation time (measured via our tokens), and the queue sizes across all satellites.

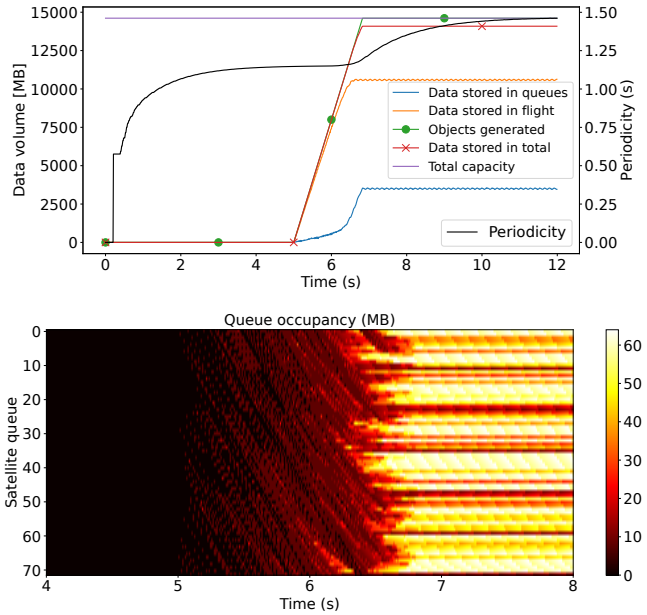

Figure 7: Evolution of storage utilization for $Q_S = 64$ MB and $S = 8$ MB in the simulated Iridium constellation: a) shows how much of the sent data is stored in queues vs. in flight as well as the evolution of the mean rotation time measured across all satellites (top); b) depicts the evolution of queue occupancy of the satellite queues used (bottom).

4.1 Basic operation

Fig. 7 shows a sample run for the Iridium constellation of 66 satellites ($N_p = 6$ orbits, $N_o = 11$ satellites, $a = 871$ km).[2] After a warmup time of 8 s, objects are generated in 1 ms intervals, the total data volume of *data generated* indicated by the green line. In the beginning, objects are usually sent immediately when they are sourced as the queues are mostly empty, shown by the steep slope of orange line representing *data stored in flight*, closely following the *objects generated*. This is confirmed by the curve for *data stored in queues* only growing gradually in the beginning. At around 6.5 s, an inflection point is reached and, statistically, all queues always contain data so that all the links are constantly busy and no more data can be stored in flight, so that further generated objects increase the share of *data stored in queues*. At around 7 s, the system is saturated and no further objects can be accepted. At this point, the theoretical system capacity (purple line at the top) is not fully reached, i.e., a fraction of objects never make it from the source queue into the system.

This effect is expected because sourcing new objects is distributed randomly across the satellites: since forwarding objects takes precedence, a newly sourced object can only enter the storage queue if there is room. Once a queue becomes full, it can only drain is there is less incoming than outgoing traffic. However, at a certain fill level, all satellites have developed standing queues (at the above inflection point) so that all links are constantly busy and queues cannot drain.

We illustrate this by looking at the queue occupancy across all satellite queues: the Northbound queues of all satellites (0–10, 12–22, ... in the plot) plus the Eastbound ones for cross-orbit connectivity (11, 23, 34, ...). Initially, only a few objects pass through the queues, which are partly filled and then emptied again. Between 6 and 7 s, we see queues grow so that standing queues develop and queue sizes largely stabilize. While there is still room in some queues to hold further objects, this space isn't available to objects sourced at other satellites. The remaining darker, i.e., less occupied, queues are the Eastbound ones because new objects contributing to queue buildup are only inserted into the Northbound ones.

4.2 Subtle Backpressure

To alleviate this statistical effect, the system needs to "move" space in the queues around to those satellites what have new objects waiting. While this could happen over time as objects expire, such an optimistic do-nothing-and-wait approach would only work proabilistically. Instead, we seek to actively make room. Since queues only drain if the incoming rate is less than the outgoing rate, we can apply some backpressure to ask the preceding satellite to reduce the outgoing rate. This must be done with care because: 1) any reduction in data rate immediately reduces the in-flight storage capacity. 2) Every satellite has a predecessor, all of which ultimately form a circle, so we must beware of cascading and oscillation effects.

[2]We use Iridium here because its constellation with just 66 satellites helps readability of the following graphs; result summaries for today's megaconstellations are included in Appendix B.

At a storage traffic rate of 80 Gbps for the ISLs, a rate change of 1% rate yields 100 MB/s and hence can assist shifting queue occupancy at short timescales. Based upon extensive simulations, we choose to limit rate adjustment $R_{adj} \in [0.0, 1.0]\%$. We define a simple feedback message sent to a neighbor satellite if the own send queue is full and (sourced) data is pending to ask the neighbor to reduce its traffic by R_{adj}. We compute R_{adj}, capped at 1 %, from the space in the send queue of an ISL, the size of waiting locally sourced objects, and the ratio of the incoming to outgoing traffic rate so that the queue drains faster than it fills until there is enough room to hold the locally sourced objects. These control messages propagate into the opposite direction and hence do not interfere with data traffic, but they could also go into a priority queue to ensure transmissioin without delay.

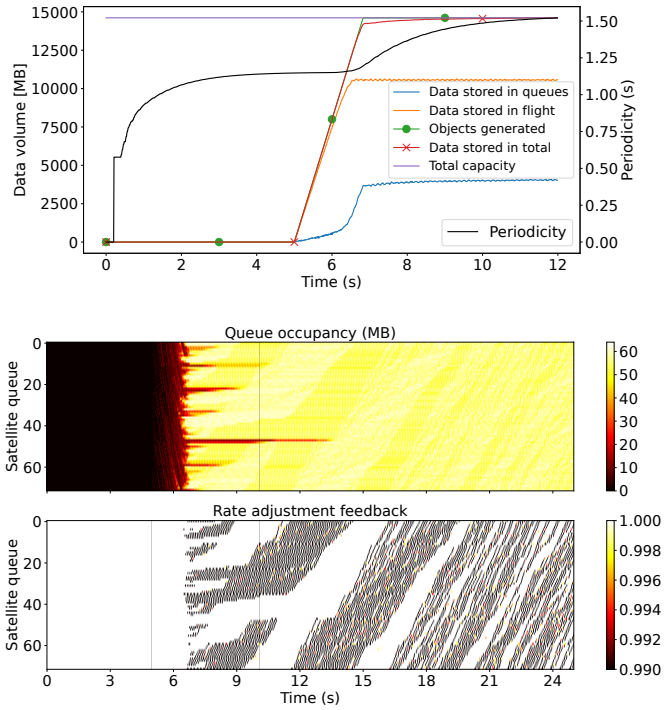

Figure 8: Applying a simple backpressure protocol helps utilizing the full storage capacity (top) and dissolve the standing queues (middle). The bottom plot shows where which backpressure is applied.

Fig. 8 depicts the effects of applying the simple backpressure protocol. The objects *stored in total* now reaches the *objects generated* (top) and the standing queues are dissolved (middle). The plot also shows the rate adjustments applied to each queue over time (bottom), indicating that the system enters and stays in a state of constant flux of minor rate adjustments. This is due to the communication latency between neighboring satellites that delays the effect of the rate adjustments: simulations without delays have shown the system to stabilize. We leave exploring proactive adjustment algorithms that can make up for the latency for future study.

The above figures show findings for the small Iridium topology to illustrate the observed effects. Appendix B also summarizes the results for the ten constellations also shown in fig. 5.

5 DISCUSSION AND CONCLUSION

In this paper, we take an unconventional approach to distributed data handling: we exploit the vast distances in space between satellites to store data "in flight" and have objects constantly rotate through all access locations so that requesting nodes just have to wait for them to pass by—rather than fixing the (replicated) storage locations and maintaining an index to find the data via a control plane. The availability of powerful ISLs in LEO satellite megaconstellations could provide a foundation for such design. While our strawman algorithm makes full use of ISLs within an orbital plane and largely spared those interconnecting different orbital planes, other configurations are conceivable, including mixing different shells to differentiate on latency and capacity needs.

But what to do with a modest amount of distributed storage capacity of some 40–80 GB at an access latency of <10s (or more with additional delay)? Anything requiring large volumes such as CDNs [4] would need different mechanisms but other interesting uses come to mind: One option is building a global user directory (a key-value store) with entries exclusively updated by their owners, e.g., along the lines of a *Minimal Global Broadcast* [10]: 128 B per user would just require 675 MB of storage and be hardly noticeable, leaving room for growth; extending this to today's 5.5 bn Internet users would require 670 GB and thus 512 MB storage queues. Another idea is using space as a backup for critical infrastructure data, such as the globale BGP routing tables: the 1.5 GB data (Sep 2024, from routeviews.org) would fit easily and need only a very small link capacity share.

While we explored storage queues of various sizes, an interesting case is an effectively queueless system for global broadcasting with memory: as shown in table 3, small content objects of a total volume of 7–100 GB could circulate once per 1–11 s. We may exploit these properties to efficiently distribute state synchronization messages in global consensus protocols where individual transactions are rather small. This may allow for a small high-priority traffic share f_s, in some cases, even LEDBAT-style scavenging traffic may suffice: both could be a start to explore this concept further.

REFERENCES

[1] J. Benet. 2014. IPFS - Content Addressed, Versioned P2P File System. arXiv.org. http://arxiv.org/abs/1407.3561

[2] Debopam Bhattacherjee, Simon Kassing, Melissa Licciardello, and Ankit Singla. 2020. In-orbit Computing: An Outlandish thought Experiment?. In *Proceedings of the 19th ACM Workshop on Hot Topics in Networks (HotNets '20)*. https://doi.org/10.1145/3422604.3425937

[3] Debopam Bhattacherjee and Ankit Singla. 2019. Network topology design at 27,000 km/hour. In *Proceedings of the 15th International Conference on Emerging Networking Experiments And Technologies (CoNEXT '19)*. https://doi.org/10.1145/3359989.3365407

[4] Rohan Bose, Saeed Fadaei, Nitinder Mohan, Mohamed Kassem, Nishanth Sastry, and Jörg Ott. 2024. It's a bird? It's a plane? It's CDN!: Investigating Content Delivery Networks in the LEO Satellite Networks Era. In *Proceedings of the 23rd ACM Workshop on Hot Topics in Networks (HotNets '24)*. https://doi.org/10.1145/3696348.3696879

[5] Bradley Denby and Brandon Lucia. 2020. Orbital Edge Computing: Nanosatellite Constellations as a New Class of Computer System. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '20)*. 16 pages. https://doi.org/10.1145/3373376.3378473

[6] Simon Kassing, Debopam Bhattacherjee, André Baptista Águas, Jens Eirik Saethre, and Ankit Singla. 2020. Exploring the "Internet from space" with Hypatia. In *Proceedings of the ACM Internet Measurement Conference (IMC '20)*. https://doi.org/10.1145/3419394.3423635
[7] Tobias Pfandzelter and David Bermbach. 2021. Edge (of the Earth) Replication: Optimizing Content Delivery in Large LEO Satellite Communication Networks. In *2021 IEEE/ACM 21st International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*. https://doi.org/10.1109/CCGrid51090.2021.00066
[8] Tobias Pfandzelter and David Bermbach. 2023. Edge Computing in Low-Earth Orbit – What Could Possibly Go Wrong?. In *Proceedings of the the 1st ACM Workshop on LEO Networking and Communication 2023 (LEO-NET '23)*. https://doi.org/10.1145/3614204.3616106
[9] F. Ross. 1986. FDDI - A tutorial. *IEEE Communications Magazine* 24, 5 (1986). https://doi.org/10.1109/MCOM.1986.1093085
[10] Christian Tschudin. 2023. Minimal Global Broadcast. Contribution to the IRTF DINRG meeting.
[11] Ruolin Xing, Mengwei Xu, Ao Zhou, Qing Li, Yiran Zhang, Feng Qian, and Shangguang Wang. 2024. Deciphering the Enigma of Satellite Computing with COTS Devices: Measurement and Analysis. In *Proceedings of the 30th Annual International Conference on Mobile Computing and Networking (ACM MobiCom '24)*. https://doi.org/10.1145/3636534.3649371
[12] Xiaolan Zhang, Giovanni Neglia, Jim Kurose, and Don Towsley. 2007. Performance Modeling of Epidemic Routing. *Computer Networks* 51, 10 (2007). https://doi.org/10.1016/j.comnet.2006.11.028
[13] Ruili Zhao, Yongyi Ran, Jiangtao Luo, and Shuangwu Chen. 2022. Towards Coverage-Aware Cooperative Video Caching in LEO Satellite Networks. In *GLOBECOM 2022 - 2022 IEEE Global Communications Conference*. https://doi.org/10.1109/GLOBECOM48099.2022.10001098

A NOTATION USED IN THIS PAPER

Table 1 summarizes the notation used in this paper.

Symbol	Explanation
r	Earth radius ($\approx$ 6371 km)
a	sallite altitude above ground
d	distance between two consecutive satellites in the same orbital plane
d_p	distance between two adjacent orbital planes at the equator
d_s	max distance between two satellites on adjacent orbital planes around the equator
N_o	number of satellite per orbital plane
N_p	number of orbital planes in a constellation
R_{ISL}	data rate of the ISL laser links
f_s	fraction of R_{ISL} allocated to storage traffic
Q_S	size of the storage queue
F_s	Data volume in flight between two satellites on the same orbit at rate $R_{ISL} \times f_S$
C_s	STorage capacity per satellite: $C_S = Q_S + F_S$
C_p	Storage capacity per orbital plane: $C_p = C_s \times N_o$
C_o	Storage capacity of the orbital shell $C_O = C_p \times N_p$
K	number of replicas per content object
C_e	effective storage capacity of the orbital shell with K replicas: $C_e = C_O/K$
c	speed of light in vacuum: $c = 299,792.458\ km/s$
t_f	processing delay per satellite: $t_f = 0.1$ ms
t_q	queuing delay per satellite: $t_q = f_s \times Q_s/R_{ISL}$
t_p	propagation delay between consective satellites within an orbit: $t_q = d/c$
t_s	propagation delay between satellites in adjacent orbits at distance d_s: $t_q = d_s/c$
t_o	rotation time of a single object through all satellites of an orbital shell
t_a	mean access time to one of K replicas of an object in a orbital shell
t_s	submission time of a newly created object to a satellite
t_e	time at which a newly created object enters the storage queue
t_w	waiting time between object creation and its entering the storage queue: $t_w = t_e - t_s$

Table 1: Overview of the symbols and notation

B DETAILED SIMULATION RESULTS

This appendix has further details on our simulations results for all the network topologies we investigated; those are summarized in table 2. As discussed above, the storage capacity of these megaconstellations is a function of the queue size per outgoing satellite link and the data in flight between any two satellites on the circular path through the orbital shell. The in-flight capacity obviously grows linearly with the number of orbits (plus marginally with the altitude) while the queue capacity grows linearly with the number of satellites; this also holds for the periodicity t_a of a data object.

Constellation	Altitude	#orbits	#sats/orbit	Inclination
S1: Starlink 1	550 km	72	22	53.0°
S2: Starlink 2	1110 km	32	50	53.8°
S3: Starlink 3	1110 km	8	50	74.0°
S4: Starlink 4	1275 km	5	75	81.0°
S5: Starlink 5	1325 km	6	75	70.0°
K1: Kuiper 1	630 km	34	34	51.9°
K2: Kuiper 2	630 km	36	36	43.0°
K3: Kuiper 3	590 km	28	28	33.0°
T1: Telesat 1	1015 km	27	13	98.98°
T2: Telesat 2	1325 km	40	33	50.88°
I: Iridium	871 km	6	11	86.4°

Table 2: LEO megaconstellation characteristics [6]

We carry out simulations using our custom simulator, written in Python, that simulates a short (stable) period in a `+grid` topology. We simulate different queue sizes $Q_s \in \{16, 32, 64, 128, 192, 256\}$ MB and explore constant objects sizes of $s \in \{1, 2, 4, 8, 16\}$ MB as well as objects sizes uniformly distributed in $s \in [1, 16]$ MB. We compute the maximum capacity C_O as per §2.2 of a (constellation, queue size) pair and then, after a warmup period ($> t_a$), fill the system up to 100% capacity, with new objects generated in 1 ms intervals at a randomly chosen satellite; objects do not expire. After capacity is reached, we run the simulation for another 10 s to determine if all objects are distributed across the storage queues.

We explore both the baseline operation (§4.1) and the simple backpressure mechanism (§4.2). To characterize the system operation, we consider two metrics: 1) The object waiting time t_w is defined as the difference between the time t_s at which the object is created at (or: submitted to) the source queue of a satellite for storage and the time t_q at which the respective object leaves the source queue and enters the storage queue and thus is effectively stored in the syste: $t_w = t_q - t_s$. Recall that stored objects take precedence over newly generated ones so that a newly created object can only enter the system if the storage queue has enough space, hence, this is a measure for the agility of the system. 2) We count n_f how many objects (if any) were not accepted into the storage system (before the end of the simulation), i.e., $t_w = \infty$, dubbed (submission) failure.

We run each simulation with ten different random seeds and report the mean number of objects created, the mean number of failure n_f, and the maximum waiting time t_w. We show the results in tables 4–14. For each message size, the upper line of the row indicates the performance of the baseline operation (§4.1), whereas the lower line shows results with backpressure (§4.2).

Across almost all simulated configuration, we find that the simple backpressure algorithm can substantially reduce the number of object submission failures by shifting queue contents around. The exception is when the queue size equals the object size, $Q_s = s = 16$ MB, in which case the message either all fit without backpressure or backpressure won't help. For the failures, note that n_f includes objects that were created close to the end of the simulation but could not be sent before the simulation ended—which may hold for some configurations with large queue sizes.

Const.	Periodicity (s)							Storage capacity (GB)						
Q_s (MB)	–	16	32	64	128	192	256	–	16	32	64	128	192	256
Starlink 1	10.9	13.6	16.4	22.0	33.1	44.2	55.3	97	122	146	196	295	394	493
Starlink 2	5.3	8.1	10.8	16.3	27.2	38.2	49.2	47	72	97	147	247	347	447
Starlink 3	1.4	2.1	2.8	4.2	6.9	9.7	12.4	12	18	24	37	62	87	112
Starlink 4	1.0	1.6	2.3	3.5	6.1	8.6	11.2	7	13	19	31	54	78	101
Starlink 5	1.2	1.9	2.7	4.2	7.3	10.3	13.4	9	16	23	37	65	93	122
Kuiper 1	5.3	7.3	9.3	13.3	21.2	29.2	37.2	46	64	83	119	191	263	335
Kuiper 2	5.6	7.8	10.0	14.5	23.5	32.4	41.3	49	69	90	130	211	292	373
Kuiper 3	4.3	5.7	7.0	9.8	15.2	20.7	26.1	38	50	62	87	136	185	234
Telesat 1	4.4	5.0	5.7	6.9	9.5	12.0	14.5	39	44	50	60	82	104	126
Telesat 2	6.8	9.0	11.3	15.9	25.0	34.1	43.3	60	81	101	142	225	307	390
Iridium	1.1	1.2	1.3	1.5	2.0	2.5	3.0	8	9	10	12	17	21	25

Table 3: Periodicities and max storage capacities for different satellite constellations and queue sizes. These theoretical values do not consider the inter-orbit links and queues, while the simulations below do, which leads a systematically lower values in this table.

Q_s	16 MB			32 MB			64 MB			128 MB			192 MB			256 MB		
s (MB)	# obj	n_f	max t_w	# obj	n_f	max t_w	# obj	n_f	max t_w	# obj	n_f	max t_w	# obj	# fails	max t_w	# obj	n_f	max t_w
1	128404	11 0	0.533 0.018	154900	808 0	1.228 0.108	207892	2995 0	0.488 0.24	313876	7467 0	0.6 0.559	419860	11859 0	0.722 4.475	525844	15942 0	0.272 1.933
2	64202	7 0	11.196 0.046	77450	611 0	0.086 0.942	103946	2063 0	0.431 1.542	156938	4674 0	1.038 4.202	209930	7057 0	0.833 4.683	262922	9376 0	3.457 5.949
4	32101	0 0	6.332 2.928	38725	543 0	1.358 6.23	51973	1592 1	0.297 9.844	78469	3279 3	1.025 12.312	104965	4714 6	0.748 12.811	131461	6038 26	1.047 17.982
8	16050	0 0	3.238 3.238	19362	368 6	1.992 11.777	25986	1157 16	0.225 10.943	39234	2260 56	0.528 13.057	52482	3152 105	2.092 15.135	65730	3975 163	0.313 15.173
16	8025	671 671	1.305 1.305	9681	50 8	9.769 10.83	12993	511 23	0.249 11.415	19617	1216 66	0.193 12.292	26241	1792 110	0.144 14.439	32865	2294 176	0.122 16.095
1,16	15106	3 3	10.822 10.347	18223	2 0	9.089 10.259	24457	29 0	12.379 11.131	36926	94 1	15.378 11.571	49395	162 4	19.435 13.195	61864	234 4	21.841 14.338

Table 4: Constellation starlink-1 storage properties

Q_s	16 MB			32 MB			64 MB			128 MB			192 MB			256 MB		
s (MB)	# obj	n_f	max t_w	# obj	n_f	max t_w	# obj	n_f	max t_w	# obj	n_f	max t_w	# obj	# fails	max t_w	# obj	n_f	max t_w
1	75496	0 0	0.0 0.0	101608	44 0	0.007 0.035	153832	732 0	0.027 0.052	258280	2831 0	0.439 0.099	362728	4979 0	0.26 0.15	467176	7037 0	0.041 0.24
2	37748	0 0	0.0 0.0	50804	84 0	0.062 0.087	76916	773 0	0.392 0.137	129140	2363 0	0.13 0.748	181364	3831 0	0.354 1.236	233588	5218 0	0.334 4.043
4	18874	0 0	0.019 0.019	25402	124 0	1.314 0.544	38458	828 0	0.147 3.013	64570	2045 0	0.122 4.082	90682	3057 0	0.156 8.987	116794	3988 0	0.238 11.35
8	9437	0 0	0.204 0.204	12701	98 0	10.16 1.36	19229	705 0	0.365 6.592	32285	1606 2	0.166 13.329	45341	2296 9	1.677 14.506	58397	2916 22	1.311 14.142
16	4718	0 0	2.186 2.186	6350	0 0	8.011 1.163	9614	334 0	0.165 9.684	16142	943 9	0.057 12.444	22670	1420 34	0.048 13.927	29198	1809 69	0.134 14.879
1,16	8881	0 0	0.497 0.579	11953	0 0	4.23 2.025	18097	16 0	12.333 2.848	30385	61 0	16.095 8.722	42673	109 0	16.986 12.553	54961	167 1	17.941 12.428

Table 5: Constellation starlink-2 storage properties

Q_s	16 MB			32 MB			64 MB			128 MB			192 MB			256 MB		
s (MB)	# obj	n_f	max t_w	# obj	n_f	max t_w	# obj	n_f	max t_w	# obj	n_f	max t_w	# obj	# fails	max t_w	# obj	n_f	max t_w
1	19944	0 0	0.0 0.0	26472	12 0	0.005 0.02	39528	188 0	0.254 0.062	65640	687 0	0.076 0.519	91752	1220 0	0.272 1.66	117864	1737 0	0.091 2.787
2	9972	0 0	0.001 0.001	13236	20 0	0.012 0.099	19764	207 0	0.393 1.287	32820	591 0	0.065 1.959	45876	961 0	0.083 3.874	58932	1285 0	0.079 4.199
4	4986	0 0	0.109 0.07	6618	35 0	0.012 0.647	9882	207 0	0.152 2.323	16410	510 0	0.778 6.031	22938	762 0	0.147 8.656	29466	995 0	0.086 11.956
8	2493	0 0	0.37 0.37	3309	32 0	10.598 4.528	4941	179 0	0.145 8.05	8205	409 3	0.137 11.544	11469	578 6	0.124 11.421	14733	732 25	0.146 11.651
16	1246	0 0	1.905 1.905	1654	0 0	3.018 2.881	2470	89 2	6.98 9.631	4102	238 7	3.001 10.912	5734	353 19	1.788 11.06	7366	458 26	2.304 11.81
1,16	2346	0 0	0.403 0.422	3114	0 0	1.266 1.65	4650	4 0	8.672 3.191	7722	21 0	10.246 10.616	10794	34 0	11.29 10.2	13866	47 1	10.78 11.527

Table 6: Constellation starlink-3 storage properties

Q_s	16 MB			32 MB			64 MB			128 MB			192 MB			256 MB		
s (MB)	# obj	n_f	max t_w	# obj	n_f	max t_w	# obj	n_f	max t_w	# obj	n_f	max t_w	# obj	# fails	max t_w	# obj	n_f	max t_w
1	15149	0 0	0.0 0.0	21229	9 0	0.033 0.005	33389	144 0	0.09 0.066	57709	570 0	0.07 0.174	82029	983 0	0.129 1.257	106349	1366 0	0.046 2.124
2	7574	0 0	0.0 0.0	10614	21 0	0.008 0.123	16694	170 0	6.248 0.588	28854	503 0	2.388 1.522	41014	819 0	4.726 3.653	53174	1094 0	8.658 4.55
4	3787	0 0	0.009 0.006	5307	29 0	0.114 0.495	8347	185 0	0.105 2.421	14427	444 0	0.075 4.679	20507	665 0	0.12 9.094	26587	866 0	0.136 9.9
8	1893	0 0	0.096 0.096	2653	26 0	0.299 2.465	4173	159 0	0.161 5.749	7213	365 0	0.121 10.021	10253	512 4	0.124 12.838	13293	646 15	0.111 12.737
16	946	0 0	0.825 0.825	1326	0 0	1.928 1.283	2086	80 0	0.111 10.757	3606	211 4	0.291 10.71	5126	316 7	0.194 11.577	6646	410 22	0.231 11.645
1,16	1782	0 0	0.239 0.258	2497	0 0	0.573 3.875	3928	4 0	10.001 2.615	6789	17 0	10.374 9.551	9650	28 0	11.603 10.405	12511	44 0	12.708 10.16

Table 7: Constellation starlink-4 storage properties

Q_s	16 MB			32 MB			64 MB			128 MB			192 MB			256 MB		
s (MB)	# obj	n_f	max t_w	# obj	n_f	max t_w	# obj	n_f	max t_w	# obj	n_f	max t_w	# obj	# fails	max t_w	# obj	n_f	max t_w
1	17992	0 0	0.0 0.0	25288	11 0	0.145 0.022	39880	185 0	0.081 0.037	69064	718 0	0.105 0.165	98248	1259 0	0.12 1.609	127432	1793 0	0.101 2.928
2	8996	0 0	0.0 0.0	12644	26 0	0.537 0.098	19940	226 0	0.321 0.586	34532	643 0	0.441 2.246	49124	1028 0	0.49 4.524	63716	1361 0	0.498 6.295
4	4498	0 0	0.016 0.016	6322	40 0	0.241 0.651	9970	230 0	0.564 4.198	17266	563 0	1.04 6.622	24562	830 1	0.974 11.755	31858	1043 2	0.985 12.347
8	2249	0 0	0.14 0.14	3161	33 0	10.318 6.01	4985	197 0	0.149 5.081	8633	446 2	0.119 10.823	12281	635 5	0.648 12.223	15929	806 17	0.163 13.011
16	1124	0 0	1.151 1.151	1580	0 0	2.422 2.39	2492	96 0	0.137 10.351	4316	261 5	0.085 10.82	6140	388 11	0.039 11.45	7964	504 33	0.04 11.646
1,16	2116	0 0	0.272 0.302	2975	0 0	7.671 0.787	4691	4 0	10.511 2.962	8125	17 0	10.724 6.426	11558	36 0	12.543 9.036	14992	52 0	11.513 10.736

Table 8: Constellation starlink-5 storage properties

Q_s	16 MB			32 MB			64 MB			128 MB			192 MB			256 MB		
s (MB)	# obj	n_f	max t_w	# obj	n_f	max t_w	# obj	n_f	max t_w	# obj	n_f	max t_w	# obj	# fails	max t_w	# obj	n_f	max t_w
1	68111	0 0	0.001 0.001	87151	136 0	0.133 0.069	125231	1036 0	0.802 0.137	201391	3165 0	5.505 0.343	277551	5217 0	0.167 0.463	353711	7264 0	1.031 0.413
2	34055	0 0	0.004 0.004	43575	157 0	0.059 0.097	62615	869 0	0.05 0.705	100695	2351 0	1.378 1.99	138775	3662 0	3.248 4.836	176855	4892 0	1.984 7.375
4	17027	0 0	0.273 0.036	21787	186 0	0.972 0.845	31307	781 0	0.291 2.657	50347	1742 0	1.074 7.666	69387	2629 0	0.803 10.843	88427	3408 0	1.127 12.662
8	8513	0 0	1.116 1.116	10893	142 0	0.969 5.928	15653	626 1	0.713 10.424	25173	1333 6	2.242 12.377	34693	1872 18	1.579 13.096	44213	2371 57	0.938 15.127
16	4256	87 87	5.876 5.876	5446	2 0	10.086 6.231	7826	288 4	0.117 10.337	12586	762 17	0.081 12.029	17346	1133 45	0.892 13.427	22106	1453 80	0.274 14.21
1,16	8013	0 0	2.376 1.978	10253	0 0	8.424 4.02	14733	16 0	10.871 9.95	23693	51 0	11.527 10.557	32653	97 1	14.55 11.268	41613	144 2	14.534 11.996

Table 9: Constellation kuiper-1 storage properties

Q_s	16 MB			32 MB			64 MB			128 MB			192 MB			256 MB		
s (MB)	# obj	n_f	max t_w	# obj	n_f	max t_w	# obj	n_f	max t_w	# obj	n_f	max t_w	# obj	# fails	max t_w	# obj	n_f	max t_w
1	73186	0 0	0.0 0.0	94498	22 0	0.003 0.012	137122	541 0	0.061 0.092	222370	2346 0	0.096 0.104	307618	4423 0	0.175 0.189	392866	6506 0	0.4 0.302
2	36593	0 0	0.0 0.0	47249	60 0	0.033 0.058	68561	635 0	2.293 0.216	111185	1988 0	2.814 1.11	153809	3264 0	1.881 2.78	196433	4547 0	1.557 3.182
4	18296	0 0	0.022 0.022	23624	88 0	0.181 0.241	34280	638 0	0.408 2.005	55592	1658 0	0.132 3.775	76904	2569 0	0.234 7.094	98216	3386 0	0.862 10.098
8	9148	0 0	0.205 0.205	11812	72 0	2.383 1.134	17140	556 0	2.13 6.866	27796	1288 1	2.048 11.237	38452	1931 6	0.951 14.895	49108	2463 14	2.141 15.543
16	4574	0 0	7.308 7.308	5906	0 0	5.347 3.118	8570	268 0	1.432 9.897	13898	769 8	0.074 11.545	19226	1175 18	1.203 13.297	24554	1509 52	0.325 14.913
1,16	8610	0 0	0.658 0.757	11117	0 0	3.83 0.373	16132	13 0	12.355 4.421	26161	51 0	16.913 9.127	36190	100 0	16.26 12.369	46219	146 0	19.686 16.164

Table 10: Constellation kuiper-2 storage properties

Q_s	16 MB			32 MB			64 MB			128 MB			192 MB			256 MB		
s (MB)	# obj	n_f	max t_w	# obj	n_f	max t_w	# obj	n_f	max t_w	# obj	n_f	max t_w	# obj	# fails	max t_w	# obj	n_f	max t_w
1	53419	0 0	0.0 0.0	66411	47 0	0.01 0.026	92395	564 0	0.058 0.114	144363	2014 0	0.112 0.265	196331	3564 0	0.13 0.479	248299	5088 0	0.173 0.924
2	26709	0 0	0.002 0.002	33205	77 0	0.047 0.088	46197	522 0	2.743 1.244	72181	1505 0	11.831 2.537	98165	2468 0	2.928 3.615	124149	3351 0	2.75 7.208
4	13354	0 0	0.357 0.053	16602	95 0	0.111 1.277	23098	495 0	0.113 3.074	36090	1189 0	0.12 4.948	49082	1789 0	0.123 10.989	62074	2371 0	0.124 10.887
8	6677	0 0	1.178 1.178	8301	79 0	5.654 5.281	11549	401 0	0.12 10.613	18045	893 3	0.16 12.983	24541	1294 14	0.172 11.997	31037	1658 29	0.124 13.205
16	3338	87 87	3.623 3.623	4150	0 0	8.034 4.083	5774	189 2	4.691 10.918	9022	507 8	2.213 11.944	12270	768 25	4.833 12.034	15518	998 51	0.164 13.175
1,16	6284	0 0	1.354 1.669	7813	0 0	4.359 3.433	10870	12 0	10.241 8.42	16983	42 0	12.143 10.922	23097	70 1	13.479 11.885	29211	101 1	13.373 13.938

Table 11: Constellation kuiper-3 storage properties

Q_s	16 MB			32 MB			64 MB			128 MB			192 MB			256 MB		
s (MB)	# obj	n_f	max t_w	# obj	n_f	max t_w	# obj	n_f	max t_w	# obj	n_f	max t_w	# obj	# fails	max t_w	# obj	n_f	max t_w
1	47635	70 0	0.611 0.778	53683	436 0	0.25 0.928	65779	1235 0	0.788 1.304	89971	2826 0	0.247 2.802	114163	4416 0	2.583 3.425	138355	6002 0	0.397 4.091
2	23817	48 0	1.764 1.601	26841	281 0	0.427 2.411	32889	724 0	0.175 3.143	44985	1582 0	0.406 4.982	57081	2398 0	0.169 6.171	69177	3197 0	0.14 9.563
4	11908	2 0	9.356 9.255	13420	214 1	0.358 10.498	16444	490 4	0.903 10.889	22492	986 16	0.615 11.746	28540	1429 31	0.791 12.878	34588	1853 48	1.014 14.381
8	5954	0 0	6.67 6.67	6710	134 18	0.937 10.623	8222	334 30	0.143 11.492	11246	626 52	0.123 12.165	14270	893 71	0.12 12.535	17294	1122 88	0.117 13.375
16	2977	364 364	0.914 0.914	3355	32 11	10.486 10.956	4111	161 22	2.42 11.156	5623	339 43	1.534 11.412	7135	481 57	0.262 11.587	8647	621 79	0.418 10.97
1,16	5604	14 5	10.225 10.547	6315	1 0	5.46 10.45	7738	11 1	10.671 7.98	10584	35 2	10.687 10.473	13431	57 2	11.322 11.315	16277	81 5	11.932 12.608

Table 12: Constellation telesat-1 storage properties

Q_s	16 MB			32 MB			64 MB			128 MB			192 MB			256 MB		
s (MB)	# obj	n_f	max t_w	# obj	n_f	max t_w	# obj	n_f	max t_w	# obj	n_f	max t_w	# obj	# fails	max t_w	# obj	n_f	max t_w
1	84993	420 0	0.303 0.466	106753	1771 0	0.105 2.54	150273	3794 0	0.074 7.201	237313	7003 0	0.292 9.045	324353	9887 0	0.563 10.067	411393	12600 1	0.118 11.593
2	42496	207 0	0.69 8.017	53376	1100 0	3.095 5.519	75136	2369 0	0.169 7.589	118656	4357 3	2.191 12.299	162176	6059 3	0.346 12.848	205696	7594 20	8.992 14.907
4	21248	101 4	10.049 10.802	26688	797 11	1.052 10.815	37568	1687 30	1.059 11.406	59328	2943 66	0.563 14.125	81088	4005 105	0.657 15.53	102848	4950 164	2.024 15.636
8	10624	5 5	10.798 10.798	13344	498 58	2.255 11.166	18784	1127 93	0.865 12.009	29664	1983 147	1.539 12.83	40544	2643 195	0.727 15.739	51424	3230 244	0.57 15.329
16	5312	543 543	1.108 1.108	6672	101 26	9.683 10.58	9392	513 45	0.208 12.382	14832	1075 91	0.142 12.659	20272	1506 134	0.107 13.95	25712	1868 192	0.095 15.031
1,16	9999	14 4	9.94 9.728	12559	6 1	10.812 10.203	17679	32 1	12.465 10.41	27919	87 2	14.808 11.058	38159	142 3	17.377 14.005	48399	193 7	18.219 14.779

Table 13: Constellation telesat-2 storage properties

Q_s	16 MB			32 MB			64 MB			128 MB			192 MB			256 MB		
s (MB)	# obj	n_f	max t_w	# obj	n_f	max t_w	# obj	n_f	max t_w	# obj	n_f	max t_w	# obj	# fails	max t_w	# obj	n_f	max t_w
1	11155	5 0	4.135 0.489	12307	54 0	0.135 0.976	14611	187 0	0.15 1.508	19219	511 0	0.262 2.721	23827	861 0	0.286 3.49	28435	1221 0	0.223 4.514
2	5577	0 0	1.831 1.154	6153	50 0	0.021 1.506	7305	137 0	0.551 2.478	9609	313 0	0.324 3.403	11913	491 0	0.21 4.693	14217	659 0	0.262 5.652
4	2788	0 0	1.178 0.986	3076	40 0	0.127 3.265	3652	91 0	0.16 6.276	4804	183 0	1.318 9.585	5956	275 0	0.563 10.046	7108	362 0	0.959 10.273
8	1394	0 0	1.327 1.327	1538	29 0	0.649 9.236	1826	63 1	0.077 10.084	2402	125 2	0.134 10.774	2978	179 7	0.045 10.157	3554	228 14	0.038 10.56
16	697	82 82	0.329 0.329	769	4 0	2.877 3.972	913	30 1	0.156 9.864	1201	63 3	2.604 10.883	1489	92 6	2.183 10.654	1777	122 12	1.645 10.31
1,16	1312	3 1	9.931 7.756	1447	0 0	6.407 1.336	1718	2 0	8.868 6.362	2261	7 0	10.799 7.903	2803	11 0	10.359 10.248	3345	16 1	10.211 10.699

Table 14: Constellation iridium storage properties