Memory-based API for real-time processing of high-data rate satellites

Adiba Firdous Nizami [1*], Md Irfan Salauddin.[2], Manikumar Vedantam.[3], Prakash Chauhan[4]

[1] Scientist/Engineer SF

[2] Scientist/Engineer SF

[3]Group Head, Satellite Real-Time Data Processing Group

[4] Director, National Remote Sensing Centre

[1,2,3,4] National Remote Sensing Centre, ISRO, India

adiba_fn@nrsc.gov.in

Abstract: *Real-time processing of satellite data necessitates pipelined parallelism wherein a block of raw-data is transformed into final-processed data block as it moves from one stage of pipeline to another. For high-data rate satellites (like Cartosat-3 with 2.88 Gbps), the rate at which the blocks-of-data is generated is very high. Hence, efficient inter-process communication (IPC) for seamless data exchange between parallel processes is necessary for real-time processing. Shared memory is the fastest IPC mechanisms but often introduce complexity due to their intricate nature. Their direct use can be challenging for developers to learn and use effectively, leading to errors and time-consuming debugging. This paper addresses this challenge by proposing a novel, memory-based read-write API specifically designed for real-time processing of satellite data. This API offers a developer-friendly, file-like interface that abstracts complex IPC mechanisms, enabling efficient data exchange. It provides flexible data-exchange between processes like that achieved using a file with comprehensive features for robust data handling such as blocking reads, timed-reading, unblocking writes to accommodate growing-satellite data acquired during ground station visibility. The API also simplifies IPC resource management with a well-defined interface and offers multiple debug modes for streamlined development. It implements abstraction for shared memory IPC handling buffer-overflow for growing satellite data thereby enhancing reliability and improving flexibility. By streamlining complex IPC tasks, this API significantly reduces development time and effort enabling developers to focus on core-algorithms while improving the overall processing efficiency of real-time satellite data systems and making it an invaluable tool for real-time processing application.*

Keywords: *Abstraction, IPC, Pipelined parallelism, Real-time Processing, Shared Memory*

Introduction

Real-time satellite data processing is crucial for applications that require immediate access to data, such as disaster response, emergency management, and other time-sensitive scenarios. Near real-time utilization of satellite data allows for faster decision-making and more effective actions during critical situations. To handle the high-data rate efficiently in real time, the satellite's ground systems and data processing infrastructure need to be equipped with scalable, low-latency inter-process communication (IPC) mechanisms, memory management strategies, and parallel/distributed computing approaches. Such infrastructure allows for rapid processing, storage, and dissemination of satellite data to applications that demand immediate access, like emergency management, defense, or environmental monitoring. Of the different components that make real-time (RT) processing possible, one critical aspect is the IPC between processes running on a processing node, as it enables seamless data exchange. This paper focuses on the importance of IPC in real-time data handling.

To ensure that high-data-rate satellite data can be processed in real time, one effective strategy is *pipelined parallelism*, where different stages of the processing pipeline handle separate blocks of data simultaneously. This approach optimizes both throughput and latency by distributing the workload across multiple stages, allowing for continuous data flow. However, for such pipelined parallelism to function effectively, *efficient IPC between stages* is essential to ensure that data is passed quickly and reliably from one process to the next. While traditional IPC mechanisms can offer the necessary speed, they are often complex, making implementation and management challenging. Therefore, the objective of this paper is to propose a memory-based API that simplifies IPC for real-time processing, enhancing both efficiency and reducing development complexity. It presents design of the library that *abstracts* the complexity of setting up and managing shared memory, synchronization mechanisms, and buffer management. By using the simple file-like read/write interface that this library provides, developers can focus on the processing logic rather than the intricate details of IPC, thereby reducing complexity and development time.

Literature Review

The increasing volume of satellite data, with advancements in high-resolution imaging satellites like Cartosat-3, has introduced significant challenges in *real-time data processing*. Cartosat-3 a third generation, agile and advanced satellite, with very high-resolution imaging capability downlinks data at rates of 2.88 Gbps. Such rates necessitate efficient real-time (RT) processing systems to handle the data volumes effectively. Research in this area predominantly focuses on distributed systems to process large datasets using scalable cloud or edge computing frameworks. However, these solutions introduce complexities and resource requirements that may not always be necessary, particularly in use cases where the data can be processed on a single node.

Many applications, such as disaster response, environmental monitoring, and defense, require satellite data to be processed and acted upon in real time. While the literature has explored real-time satellite data processing in distributed systems, there is limited research on how single-node systems can meet these real-time demands without the overhead of distributed frameworks. In cases where low-latency processing on dedicated single-node systems is sufficient, optimizing the data exchange between parallel processes becomes crucial.

Inter-process communication (IPC) is a core component of real-time processing systems, especially for parallel processes running on a single node. Traditional IPC mechanisms, including pipes, message queues, and shared memory, have been widely studied for their role in facilitating communication between processes within the same machine. While shared memory offers the advantage of low-latency data exchange, its implementation and management are complex especially in high-throughput, real-time environments.

Research on IPC has often focused on their use in distributed systems, where the emphasis is on managing communication between nodes over a network. However, in the context of parallel processing on single-node, there is limited discussion on simplifying IPC for real-time performance without distributed processing. Studies that do address IPC on single node environments frequently highlight the difficulties of synchronization, memory management, and resource contention. These challenges make traditional IPC usage cumbersome, especially in real-time satellite data processing applications where efficiency and simplicity are crucial.

Memory-based IPC, particularly through shared memory is an efficient method for high-speed communication between parallel processes on a single node. However, traditional shared memory implementations require developers to manually manage memory allocation, synchronization (e.g., using semaphores or mutexes), and avoid race conditions. There is a clear need for solutions that abstract these low-level operations, enabling developers to focus on application logic rather than the intricacies of memory management.

Pipelined parallelism is a widely used approach for optimizing data processing throughput and latency in real-time systems. By dividing the data processing workflow into distinct stages, each stage can operate on separate blocks of data in parallel, ensuring continuous data flow. This architecture has been particularly effective in high-performance computing (HPC) environments and real-time systems, where it allows for the concurrent processing of data streams. However, efficient data exchange between stages in a pipelined parallel system relies heavily on the underlying IPC mechanism. Therefore, an optimized, simplified IPC approach that can handle high-throughput data exchange in a pipelined architecture is needed for real-time satellite data processing on a single node.

Libraries like POSIX shared memory and System V IPC offer standardized approaches to managing shared memory, message queues, and semaphores. However, these libraries still require significant expertise to implement and manage effectively in real-time systems. Recent research has explored higher-level *abstractions* and APIs designed to simplify IPC, but these solutions often target distributed computing environments and do not fully address the specific challenges faced by single-node, real-time systems. This paper proposes an API that simplifies IPC by offering a file-like read/write interface, abstracting the complexity of traditional shared memory implementations. The proposed API is well-suited for pipelined parallel processing, providing a streamlined approach to data exchange between parallel processes running on the same machine.

Methodology

Design of the library

The core requirement for the proposed library was to facilitate efficient data exchange between parallel processes using a simplified interface. Specifically, Process 1 writes M bytes of data (of datatype1) into shared memory using the APIs mwrite() call, while Process 2 reads N bytes of data (of datatype1) using the APIs mread() call. Importantly, M and N can be different, and the library should ensure that if Process 2 requests more data than is available (N > M), then it automatically makes Process 2 wait until the required amount of data is available. This feature provides flexibility in data handling to both processes 1 and 2 while abstracting the complexities of internal memory shared between them and the synchronization required.

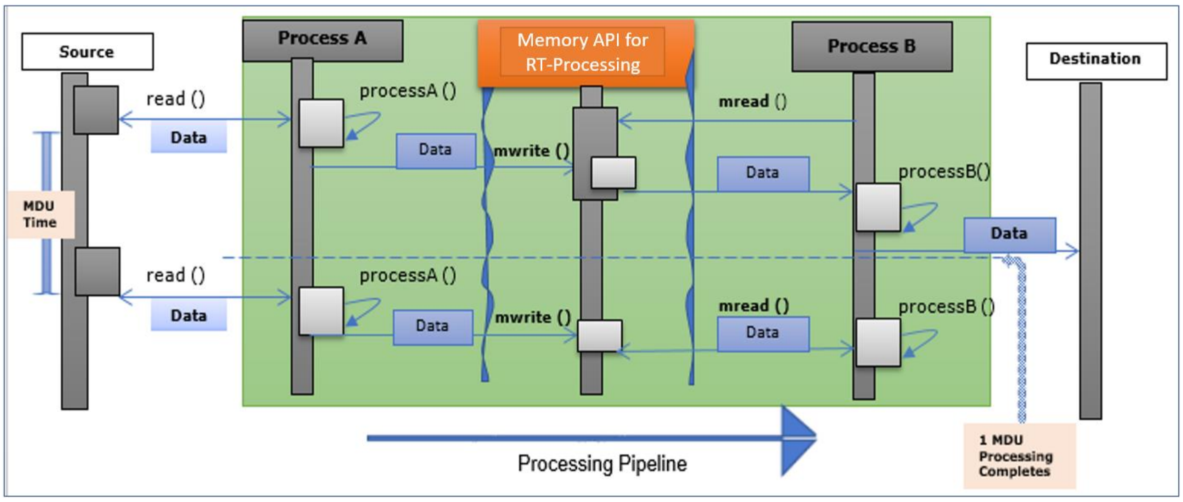


Figure 1: Typical sequence of library/API calls

Design Considerations

Given the constraints and requirements of real-time processing on RAM-limited systems, the following design decisions were made to optimize inter-process communication using System V shared memory IPC:

- *Shared memory implemented as circular-buffer:* To efficiently handle the memory limitations of the system, shared memory is implemented as a circular buffer. This approach allows for continuous writing and reading of data without exceeding the memory constraints, even when high volumes of data are being processed.

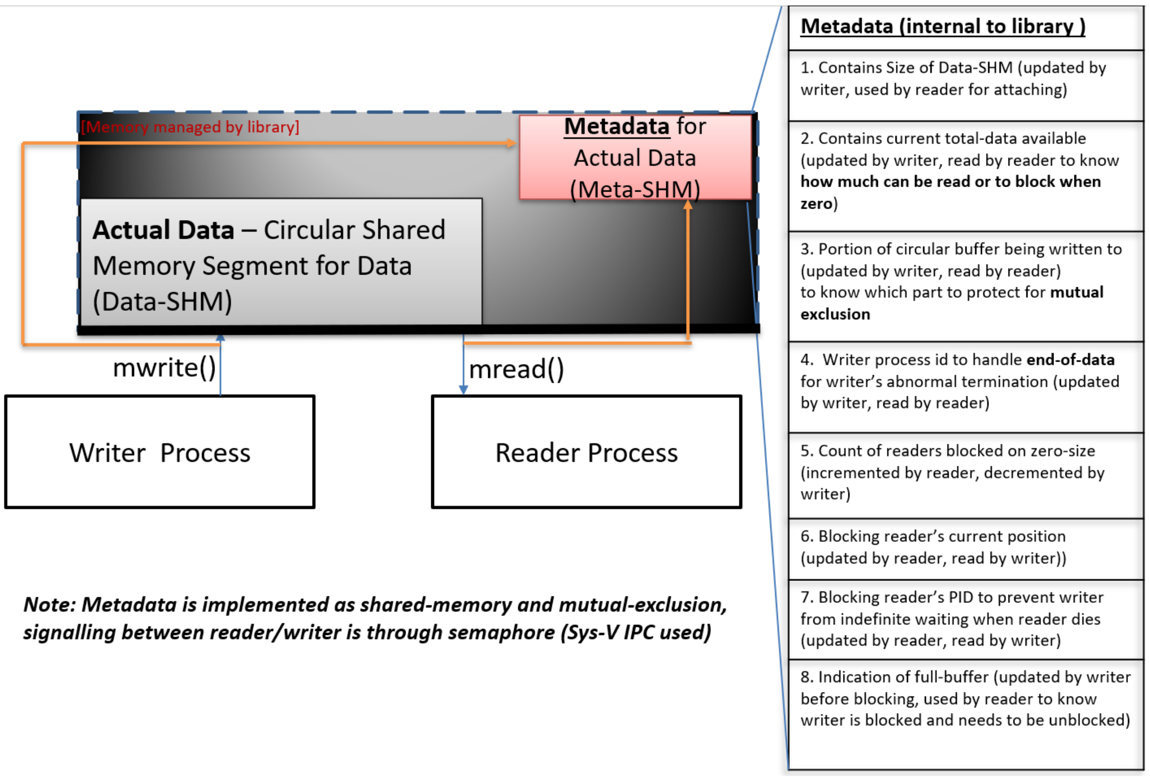
- *Synchronization of shared memory*: The circular buffer requires explicit synchronization between the writer (Process 1) and the reader (Process 2) to ensure mutual exclusion. System-V semaphore is used by library for this purpose to prevent race condition. The semaphore is employed to protect specific portions of shared memory being accessed. This ensures that while one process is writing, the other process cannot access the same memory segment, thus preventing data corruption
  When reading process (say Process 2) attempts to read more data than is currently available in the shared memory, it is blocked until the required data becomes available. This blocking is managed by a semaphore that signals when sufficient data has been written (say by Process 1)
- *Variable Read/Write block-size in circular Buffer:* The amount of data being written by a process (say process 1) may be M bytes every time but for the last buffer being processed, the number of bytes may be less than M. Similarly, the reading process (say process 2) may request N bytes for read. So, the size of block being written to or read from circular buffer for a given data type is variable i.e. not fixed.
- *End-of-Data Indication for Reader*
  Termination Handling: The library includes mechanisms to indicate both normal and abnormal termination of the writing process so that reading process (Process 2) is aware of whether more data can be expected or if the writer has terminated.
  Data Availability: A metadata structure is maintained within a separate shared memory segment. This metadata tracks the total number of bytes written so far and is accessible to both the writer and the reader.
- *Fast Writer and Slow Reader Scenario*
  Buffer Overflow Detection & Prevention: To avoid the writer overwriting unread data in the buffer, the library provides feature of blocking the writer (Process 1) when the buffer is full using semaphore. This blocking is temporary until the reader consumes enough data to free up space. Also, to prevent infinite blocking of the writer in cases where the reader is significantly slower or unresponsive, time-out is present for blocking. However, if data is indeed lost due to a very slow reader, the library ensures the next read after buffer-miss is handled gracefully.

Features of the library

1. Simple file-write like write call to write requested size of data to memory

2. Simple file-read like read call to read requested size of data from memory with blocking until requested size available
3. Timed reading implemented to prevent indefinite blocking for data availability
4. Data availability signalling by writer to readers blocked on requested data size
5. Creation, initialization and clean-up of IPC resources through simple functions
6. Debug modes with option to direct print to user provided log file
7. Library also provide write-to-file option which when enabled writes the data written to memory to a user-provide file

Implementation Details


[Memory managed by library]
Metadata for Actual Data (Meta-SHM)
Actual Data – Circular Shared Memory Segment for Data (Data-SHM)
mwrite()
mread()
Writer Process
Reader Process

Note: Metadata is implemented as shared-memory and mutual-exclusion, signalling between reader/writer is through semaphore (Sys-V IPC used)

Metadata (internal to library)

1. Contains Size of Data-SHM (updated by writer, used by reader for attaching)
2. Contains current total-data available (updated by writer, read by reader to know **how much can be read or to block when zero**)
3. Portion of circular buffer being written to (updated by writer, read by reader) to know which part to protect for **mutual exclusion**
4. Writer process id to handle **end-of-data** for writer's abnormal termination (updated by writer, read by reader)
5. Count of readers blocked on zero-size (incremented by reader, decremented by writer)
6. Blocking reader's current position (updated by reader, read by writer))
7. Blocking reader's PID to prevent writer from indefinite waiting when reader dies (updated by reader, read by writer)
8. Indication of full-buffer (updated by writer before blocking, used by reader to know writer is blocked and needs to be unblocked)

Figure 2: Internals of the data managed by library

Memory Write Interface

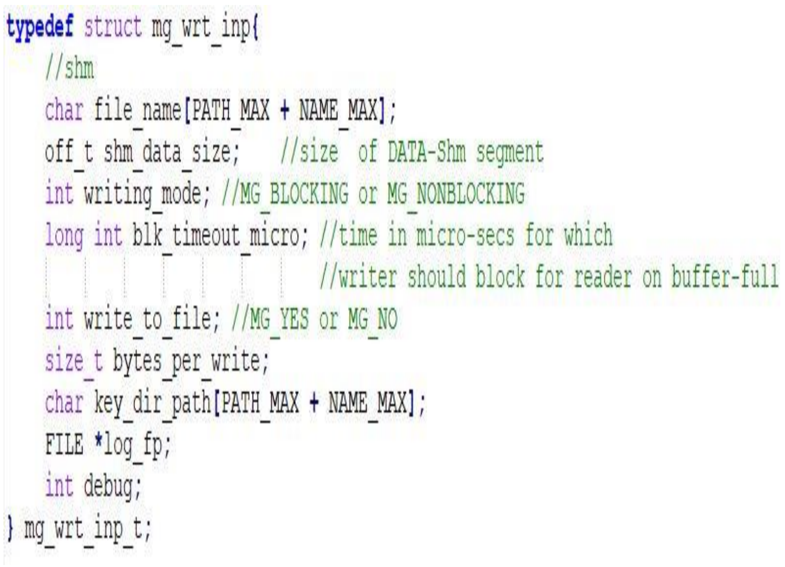
```
int mg_write_init(mg_wrt_inp_t *write_inp);
ssize_t mg_write(mg_wrt_inp_t *write_inp, const void *buf, size_t count);
int mg_write_cleanup(mg_wrt_inp_t *write_inp);
```

Memory Read Interface

```
int mg_read_init(mg_read_inp_t *read_inp);
ssize_t mg_read(mg_read_inp_t *read_inp, void *buf, size_t count);
int mg_read_cleanup(mg_read_inp_t *read_inp);
```

Figure 3: read/write API

Inputs from user of the library



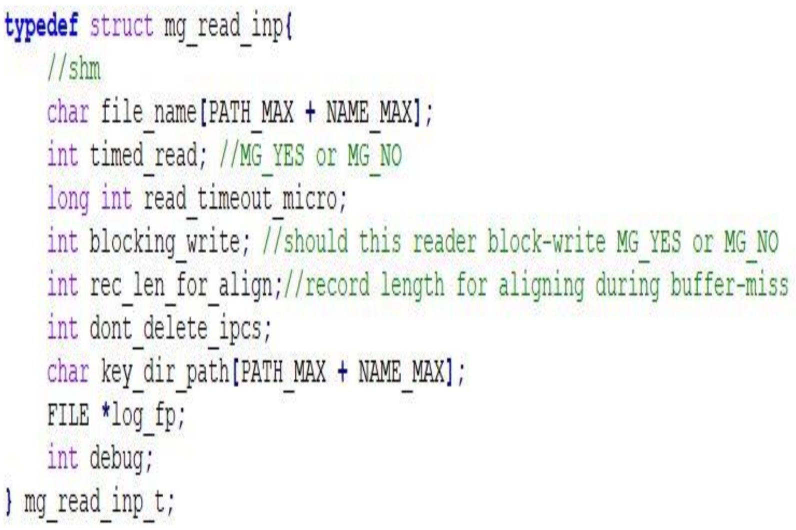
```
typedef struct mg_wrt_inp{
    //shm
    char file_name[PATH_MAX + NAME_MAX];
    off_t shm_data_size;    //size  of DATA-Shm segment
    int writing_mode; //MG_BLOCKING or MG_NONBLOCKING
    long int blk_timeout_micro; //time in micro-secs for which
                                //writer should block for reader on buffer-full
    int write_to_file; //MG_YES or MG_NO
    size_t bytes_per_write;
    char key_dir_path[PATH_MAX + NAME_MAX];
    FILE *log_fp;
    int debug;
} mg_wrt_inp_t;
```

Inputs to be passed by writer

- Data ID = Absolute path of (non-existing) file representing data to be written
- Size of DATA-Shared memory segment (circular buffer) to be created
- Writer-blocking-mode : enable/disable (default) block of writer on buffer full
- Maximum time-out for blocking if writer is in blocking mode

Figure 4: Input structure to be passed by writing process

- Enable write of data to file, IPC Keys directory, Log File Pointer, Debug Mode



```
typedef struct mg_read_inp{
    //shm
    char file_name[PATH_MAX + NAME_MAX];
    int timed_read; //MG_YES or MG_NO
    long int read_timeout_micro;
    int blocking_write; //should this reader block-write MG_YES or MG_NO
    int rec_len_for_align;//record length for aligning during buffer-miss
    int dont_delete_ipcs;
    char key_dir_path[PATH_MAX + NAME_MAX];
    FILE *log_fp;
    int debug;
} mg_read_inp_t;
```

Inputs to be passed by reader

- Data ID = Absolute path of (non-existing) file representing data to be written
- Timed-read = enable time-out if requested data unavailable until
- Time-out in microsecs for timed-read
- Whether this reader should block writer

Figure 5: Input structure to be passed by reading process

- In case of buffer-miss, the record length to be used for aligning next read
- Don’t-delete IPCS Flag – If set, underlying IPCs are NOT deleted on cleanup else IPCs are deleted (default)
- IPC Keys directory, Log File Pointer, Debug Mode

Details of API functions

Table 1: Details of API functions

mq_write_init(&write_inp): • Validates inputs passed by user • Create and attach IPCs i.e. semaphore and metadata-shared memory segment *(meta-shm)* and attach the meta-shm to calling process (writer) • Create shared-memory segment for data *(data-shm)* with user-provided circular buffer size (segment-size), update this segment-size in meta-shm and attach data-shm to calling process (writer) • Create zero-sized data-ID-file corresponding to user passed data-ID • If write_to_file enabled, initialize writer-threads thread-pool, open the data-ID-file(fd) for writing	***mq_read_init(&read_inp):*** • Validates inputs passed by user • Busy-wait for availability of the data-ID-file passed by user • Get & attach IPCs structures i.e., semaphore, meta-data segment to calling process (reader) • Get the size of data shared-memory segment (data-shm) from meta-data segment (meta-shm) and attach the data-shm to calling process (reader) • If this reader blocks writer, update the reader process id (reader-pid) in meta-data segment (meta-shm)
mq_write(&write_inp, buf, count) • If write in "*BLOCKING*" mode i.e., blocks until reader reads, *BLOCK* for reader to read when buffer is full until space created or reader (reader-pid) terminates • Announce the portion of circular buffer about to be written for readers in meta-shm • Write "count" bytes of user provided data "buf" to data-shm circular shared-memory buffer • Update the total size (no. of bytes) available in data-shm circular buffer in meta-data segment • *UNBLOCK* readers blocked on zero-size • If write_to_file enabled by user, issue write to a thread in *pool*	***mq_read(&read_inp, buf, count)*** • Get the total-size of data available from meta-shm • *BLOCK* on zero-size • Before reading, check if any data-missed (i.e., overwritten) • If missed, print number of bytes lost and read from a position ensuring read is record-aligned • read "count" bytes from the buffer based on data-availability. if request "count" bytes available, read and return if less than "count" bytes available, wait until ". complete" file found, writer-terminates or time-out occurs (for timed read) • If this reader blocks writer, *UNBLOCK* writer when required space created in buffer for writing

mq_write_cleanup(&write_inp)	***mq_read_cleanup(&read_inp)***
• If *write_to_file* enabled, do thread-pool cleanup and close the data-ID-file (fd), • detach meta-shm and data-shm shared-memory segments	• detach meta-shm and data-shm shared-memory segments • delete IPCs except when don't-delete flag set by user

IPC Keys Management

The library uses System-V IPCs i.e., shared memory to implement the actual data sharing between processes, shared-memory for metadata and semaphore for mutual exclusion and signalling. System V IPCs require keys. To have generic mission-independent way of creating keys for IPC, data ID (See figures 4 and 5) requested by library users are unique file-paths which are used for IPC key-generation (using ftok) as follows: a) *"key_dir_path"* user input in read/write input structures must be an existing path unique for real-time processing instance b) *Base name of "file_name"* in read/write input which represents absolute path of the data-file . "key_dir_path" + "base_name(file_name)" is used to create 3 distinct paths to create 3 IPC keys per file - 1 for Metadata Shared Memory Segment, 1 for Semaphore and 1 for Actual data Shared Memory. These key values are stored in "key_dir_path" directory for reader-writer to agree on same key value and for subsequent clean-up. All processes must pass same "key_dir_path" so that cleanup can be done from single path.

Testing: The library is implemented in C language and deployed as shared library. It was tested for correctness, dynamic scenarios and performance. Correctness means to establish that reading process has read the data exactly as written by writer. To check for correctness, test writer code was developed which writes the same data to file Wfile while writing it to memory using mwrite() and reader also writes the data read using mread() to file Rfile. The files Wfile and Rfile are compared to check if they are same. This *correctness* was tested with multiple test cases viz., reader and writer requested read-size and write-size, different read-size and write-size, read-size/write-size factor of internal circular buffer, read-size/write-size not factor of internal circular buffer to capture boundary conditions. *Dynamic scenarios* such as slow reader-fast writer, slow writer-fast reader were also tested by putting random amount of sleep between successive read/write calls. This was to check whether buffer full condition was encountered by reader, whether writer (blocked) waited for reader to read on buffer full (when blocking is enabled). If writer was set non-blocking and buffer-overflow occurred, bytes-missed to overflow are printed for reader and next-read is aligned to correct position. The library is only meant for simplified data-exchange between 2 parallel processes. As such, it does not do any processing. So, testing the performance of library functions meant ensuring that the read/write to memory does not significantly contribute to execution time of the calling process. This was tested using 2-parallel processes– 1 writer and 1 reader running together in pipeline: writer invoked write call iteratively with a sleep of N (5.12ms) milliseconds, similarly reader invoked read call iteratively with a sleep of N milliseconds. Duration for Sleep + memory-write for write/ duration for memory-read + sleep duration was measured for each iteration and average duration was measured. This average was slightly more than sleep duration N (varied from 5.2 to 5.4 milliseconds). These test scenarios are ideal simulated scenarios for the propose of testing. Practical testing scenario (See results/discussion) is when all the processes involved in RT-processing are integrated and use library for data-exchange and the end time of last process in the pipeline closely matches with the end-time of the last data block pumped in the pipeline.

Results and Discussion

This library was successfully demonstrated for RT pre-processing of Cartosat-3. Cartosat-3 is a third generation of cartographic imaging satellite, weighing around 1500 kg and launched into 505 km Polar Sun Synchronous orbit in Nov 2019 by Indian Space Research Organization. It has a panchromatic resolution of ~0.28 meter and multi-spectral resolution of ~1.13 meter which is a major improvement from the previous payloads in the Cartosat-2 series. Potential uses include weather mapping, cartography, disaster and other critical applications. Panchromatic (PAN) payload data rate is around 18Gbps and multi-spectral (MX) payload data rate is of 4.5Gbps. The total payload data rate is around 22.5 Gbps. Data transmission from satellite is either in X-Band or Ka- Band. In case of X-Band, data is transmitted in two streams while in Ka band, data is transmitted in six streams. Each stream downlink rate is 480 Mbps. Therefore X-band downlink rate is 960 Mbps, whereas Ka- Band downlink rate is 2.88 Gbps.

Data rate and Data Volume for Ka band acquisition:

- Average Dumping Duration for Ka – Band = 5 min = 300 seconds
- Data Rate (Ka Band): 480 Mbps * 6 Streams ~ 400Mbps * 6 = 2.4 Gbps = 300MBps;
- Acquired Data Volume: Raw data per pass: (300 MBps * 300 sec)/ 8 = 90GB

- Processed and decompressed data per pass (nominal compression ratio is 5) = (90 * 5) =450GB

Six streams of Cartosat-3 data are acquired and processed in real time on identified high-end server to generate Level-0 products (i.e., Level-0 Raw data) for each of the sensors. The RT-processing is done as part of RT-processing framework which comprises of scheduler (that instantiates & monitors all configured RT-processes), memory -library for data exchange between processes and various configuration files. The memory-library forms the core of the framework that enables RT-data exchange. This section describes the payload data pre-processing pipeline realized using the library. The payload data pre-processing pipeline to generate Level-0 products comprises of following processes (stages) – Data Acquisition, RSDecodingN CADUDataExtraction (CADUExt), SpacePacketsExtraction (APIDExt), Data Align-Merge & Archival (DAMAR), Decompression, SubsampledVideoDataExtraction (SVDExt) and QuickLookDisplay.

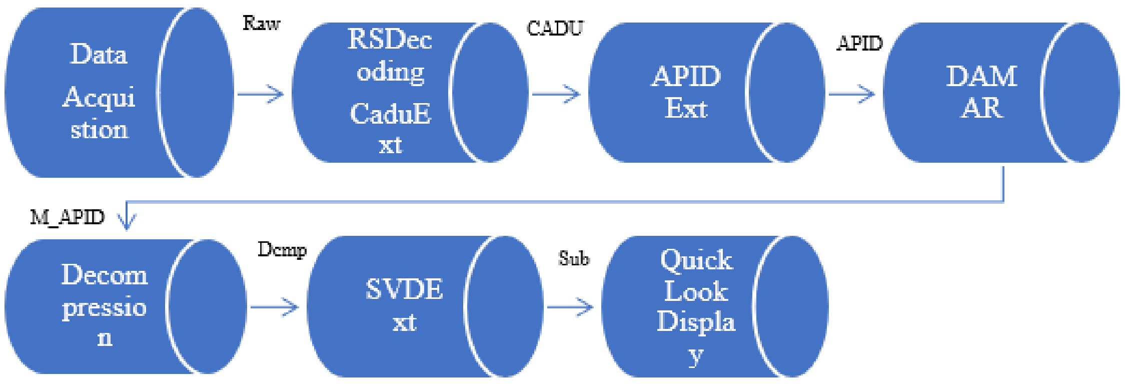


Figure 6: Cartosat-3 Payload data pre-processing pipeline

The data types and no. of instances of each type involved in the above pipeline for Cartosat3:

- Raw=Raw data as acquired from hardware = 6 no.
- CADU=Channel Access Data Unit data extracted from Decoded data = 16 (8 PAN, 8 MX)
- APID=Application Process Identifier space packets data extracted from CADU= 120 (24 PAN, 96 MX)
- M_APID=Merged and aligned APID data= 5 (1 PAN, 4 MX)

- Dcmp= Decompressed data = 125 (120 APID-wise PAN + MX & 1 PAN, 4 MX)
- Sub=Subsampled data = 5 (1 PAN, 4 MX)

Total data shared memory (data-shm) segments involved for preprocessing = 6 + 16 + 120 + 5 + 125 + 5 = 277. For each data-shm, there is one meta-shm and one semaphore set used.

Sizing of data-shm segments: Each type of data except decompressed & subsampled data is generated at the downlink rate of 300MBps. Decompressed data is generated at 5 times the download rate and subsampled at 1/4th of decompressed data rate. Depending on the data-type, the data-shared memory segment size is configured large enough to hold data for duration sufficient for next-reader process to consume it.

- Total Raw data ~2min = 2x60x300 = 36 GB
- Total CADU data ~30secs = 30x300 = 9GB
- Total APID data ~30 secs = 30x300 = 9GB
- Total Merged APID data ~ 2min = 2x60x300 = 36GB
- Decompressed data ~30 secs = 30x300x5 =45GB
- Subsampled data ~30 secs = (30x300x5)/4 ~ 11.25GB

Raw data and merged-APID data sizes are allocated so as to buffer upto 2-minutes of data as the subsequent process i.e., RSDecodeCADUExt and Decompression are compute-intensive and may have variability in processing time depending on data. Total shared-memory size allotted for data-shm segments = 146.25GB

Test system configuration: Server used for RT-processing - Intel Xeon(R) Gold 6154 @3.0Ghz, 4 CPU 18 Core (144 logical cores) server with 512 GB of memory & Red Hat Enterprise Linux version 7.5

With the above configurations, for an average pass duration of 5 mins, best case RT-preprocessing time was within 30 seconds from LOS and worst case was 2 mins from LOS (loss of signal).

Limitations/Constraints: While the memory-based API simplifies and optimizes inter-process communication (IPC) through efficient shared memory management, the overall success of real-time (RT) processing is highly dependent on two additional factors:

- Efficiency of Process Implementation: The performance of RT-processing is significantly influenced by how well each individual process in the pipeline is coded. Optimized algorithms, efficient use of memory and CPU resources, and minimizing latency in data-handling routines are critical. Poorly optimized processes can become bottlenecks in the pipeline, undermining the benefits of the memory-based IPC library.

- Server Configuration: The computational capacity and configuration of the server hosting these processes also play a pivotal role. RT-processing of high-data-rate satellite data demands high-end servers with sufficient CPU cores, RAM, and fast storage to ensure smooth operation. Without a powerful server configuration, even an optimized library cannot fully meet the stringent demands of real-time processing, particularly when dealing with high data rates such as those generated by satellites like Cartosat-3.

Thus, while the library abstracts IPC complexity, ensuring real-time performance depends on both the *quality of process coding* and the *hardware infrastructure* on which the system operates.

Conclusion and Recommendation

The proposed memory-based API simplifies and optimizes inter-process communication (IPC) for real-time processing of high-data-rate satellite data. By abstracting complex System V IPC mechanisms into straightforward read/write operations, the library reduces development complexity, allowing developers to focus on core processing tasks. The use of shared memory as a circular buffer and semaphores for synchronization ensures efficient and reliable data exchange between parallel processes. The library also accommodates variable data block sizes, providing flexibility in handling diverse satellite data formats. However, while the library significantly streamlines IPC, the overall performance of real-time processing is heavily dependent on how well individual processes are optimized and the computational resources of the server used for processing. To ensure the library’s robustness and efficiency in diverse real-world scenarios, comprehensive testing and benchmarking should be performed. This would allow for better understanding of the library's limits under different processing loads and data rates.

References

Daryl A. Swade and James F. Rose. OPUS: A flexible pipeline data-processing environment. In Proceedings of the AIAA/USU Conference on Small Satellites. September 1998.

Downey, A. (2016). *The little book of semaphores* (2nd ed.). Green Tea Press. https://greenteapress.com/semaphores/LittleBookOfSemaphores.pdf

Indian Space Research Organization. (2019). *PSLV-C47 launch kit*. https://www.isro.gov.in/media_isro/pdf/Missions/PSLVC47/PSLV_C47LaunchKit_cdr.pdf

JáJá, J. (1992). *An introduction to parallel algorithms*. Addison Wesley Longman Publishing Co., Inc.

Patterson, D. (n.d.). *Pattern: Barrier synchronization*. Berkeley EECS Patterns. https://patterns.eecs.berkeley.edu/?page_id=542

Raj, R. (n.d.). *Software design: What is abstraction?* Spring Boot Tutorial. https://www.springboottutorial.com/software-design-what-is-abstraction

Shamsudeen, D. (2018). A Study on Inter process Communications in Distributed Computer systems.

Stevens, W. R. (1999). UNIX network programming. Volume 2, Interprocess communications (2nd edition). Addison Wesley Longman Singapore Pte. Ltd.

The New Stack. (2021, May 5). *Why your code needs abstraction layers*. The New Stack. https://thenewstack.io/why-your-code-needs-abstraction-layers/

X. Sun, B. Li, T. Shi, Y. Hu, X. Yang and Y. Song, "Real-time Processing for Remote Sensing Satellite Data Based on Stream Computing," *2019 IEEE International Conference on Signal, Information and Data Processing (ICSIDP)*, Chongqing, China, 2019, pp. 1-8, doi: 10.1109/ICSIDP47821.2019.9173437.

Zhang, Yang & Yu, Dan & Ma, Shilong. (2011). Researches on real-time satellite data flow processing based on file buffer. 3. 1768-1774. 10.1109/ICNC.2011.6022348.