Fast Random Access Log (FRAL)

[MS Thesis Defense]

David Maaghul

Outline

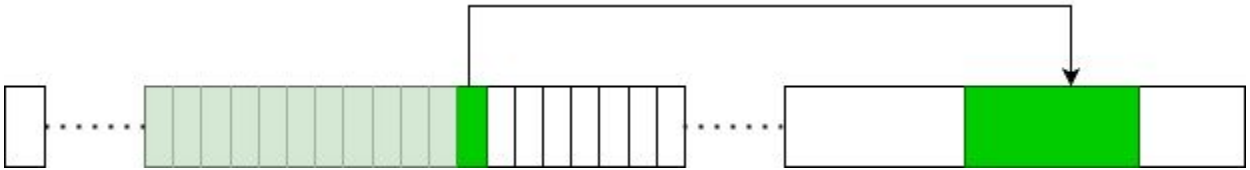
- Overview
- Core Architecture
- Performance Testing: C++ and Python
- Simple Sample Usage
- 3 Advanced Use Cases
- Demo
- Future Work

Overview

- Want to create data structure that allows multiple processes to read from and write to a shared array-indexed log as fast as possible
- Design Considerations:
    - Log is allocated over shared memory
        - Our current implementation uses memory-mapped files
    - Non-blocking writes
    - Reads are random access
- Implementation details:
    - Core engine implemented in C++
    - Python bindings provided and tested for convenient use (with less performance constraints)

Core Architecture

- The interface is very small and simple
- Three main (non-blocking) functions:
  - **allocate:** allocates n contiguous bytes of the memory space (a blob) and returns a pointer to the beginning of the allocation. This is similar to malloc in C.
  - **append:** appends the address of a memory allocation (blob) to an indexed log, updates the size of the log and returns the index of the appended blob.
  - **load:** provided a log-entry index, returns the address of the corresponding memory allocation
- There is no **store** function; appends are permanent but blobs can change
- Virtual Memory = different addressing for different processes
  - Store offset from beginning of shared-memory allocation to array
  - Offset corresponds to array index = random access



Non-Blocking Allocate

- Our first atomic variable:
  - *h*: the location of the next free memory address
- Algorithm uses fetch-add primitive
  - Offset of next free memory address is an atomic variable
- Caveat:
  - Process failure (e.g., the OS kills the process) after fetch-add step (line 5) results in wasted space.
    - Still structurally consistent

Algorithm 1 Non-Blocking Allocate

```
1: s: allocation size (bytes)
2: S: size of memory space (bytes)
3: h: offset of next free memory address
4: function ALLOCATE(s)
5:     c = FetchAdd(h, s)
6:     if c + s > S then
7:         return null
8:     end if
9:     return c
10: end function
```

Non-Blocking Append

- Motivation for approach:
  - How we expect log tailing to be implemented is critical in our design of our append algorithm

Algorithm 2 Tail Log

S: max size of log
$fral$: the random access log
function READ($fral$)
    $i \leftarrow 0$
    for $i < S$ **do**
        $blob$ = load($fral$, i)
        if $blob$ is not null **then**
            ProcessBlob($blob$)
            $i \leftarrow i + 1$
        end if
    end for
end function

Non-Blocking Append (Continued)

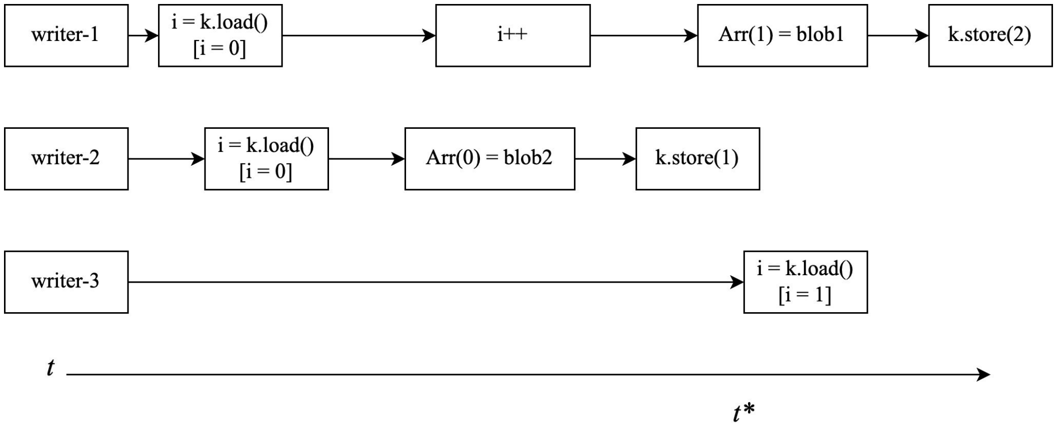
- Two new atomic variables:
  - *Arr[i]*: The i^{th} entry of the array of offsets
  - *k:* The next available array index
- Use compare-and-swap primitive:
  - Only store offset value if corresponding entry is available
- A process can fail at any point during the append
  - Array entries are never wasted
  - Iterating over log will never yield infinite loop (previous slide)
- In the case of *n* writers in parallel:
  - A given writer has a worst case operation count of *O(n)* and the total operation count between the *n* writers is $O(n^2)$
  - We could achieve this with *O(1)* and *O(n)*, respectively, but would then lose the promise of not wasting array entries
    - Asymptotic gain didn't materialize in performance testing

Algorithm 3 Non-Blocking Append

```
blob: allocation to be appended
S: max size of log
k: the next available index
a: address of start of shared memory space
Arr: the log of offsets
function APPEND(blob)
    z ← BlobOffset(a, blob)
    i ← AtomicLoad(k)
    for i < S do
        if CompareAndSwap(Arr[i], 0, z) then
            AtomicStore(k, i + 1)
            return i
        end if
        i ← i + 1
    end for
    return -1
end function
```

Non-Blocking Append (Continued)

- Problem: can't atomically update two variables at the same time
- The next available index is estimated (relaxed):
    - Example with 3 writers:
        - writer-3 loads k = 1 at time *t**, but entry 1 has already been appended to by writer-1
        - writer-3 needs two compare-and-swap operations to realize and write to the next available index (k = 2)

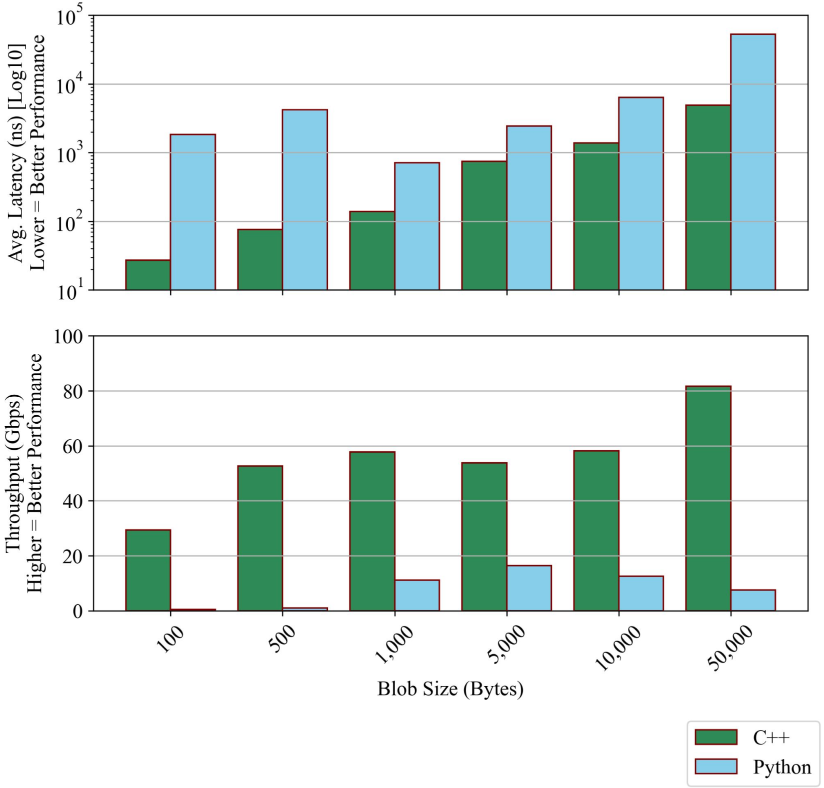


Performance Testing Results

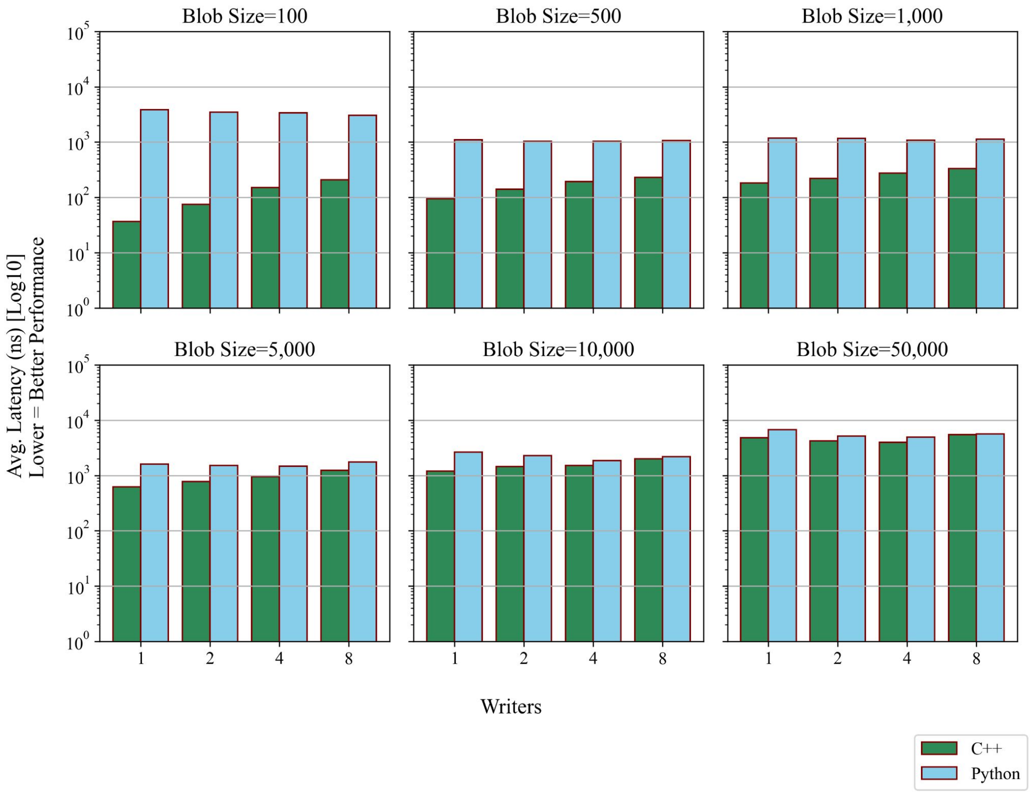
- ***Two tests of core architecture performance:***
  - Look to measure throughput and latency in various ways
  - *Write Test:*
    - Measures the speed and efficiency of writing *N* GB of data to the log in *n*-byte blobs
    - Assesses performance of high volume of write operations by one writer
    - The timer starts before the first write and ends after the last; latency is an average
  - *Producer-Consumer Test:*
    - Measures the speed and efficiency of simultaneously writing and reading *N* GB of data to the log in *n*-bytes blobs with *k* writers
    - Assesses performance of concurrent read and write operations from multiple sources
    - The timer starts before the first write and ends after the last read; latency, again, is an average

*Note that all testing was done with an 8-core Apple M2 machine with 8GB of unified RAM.

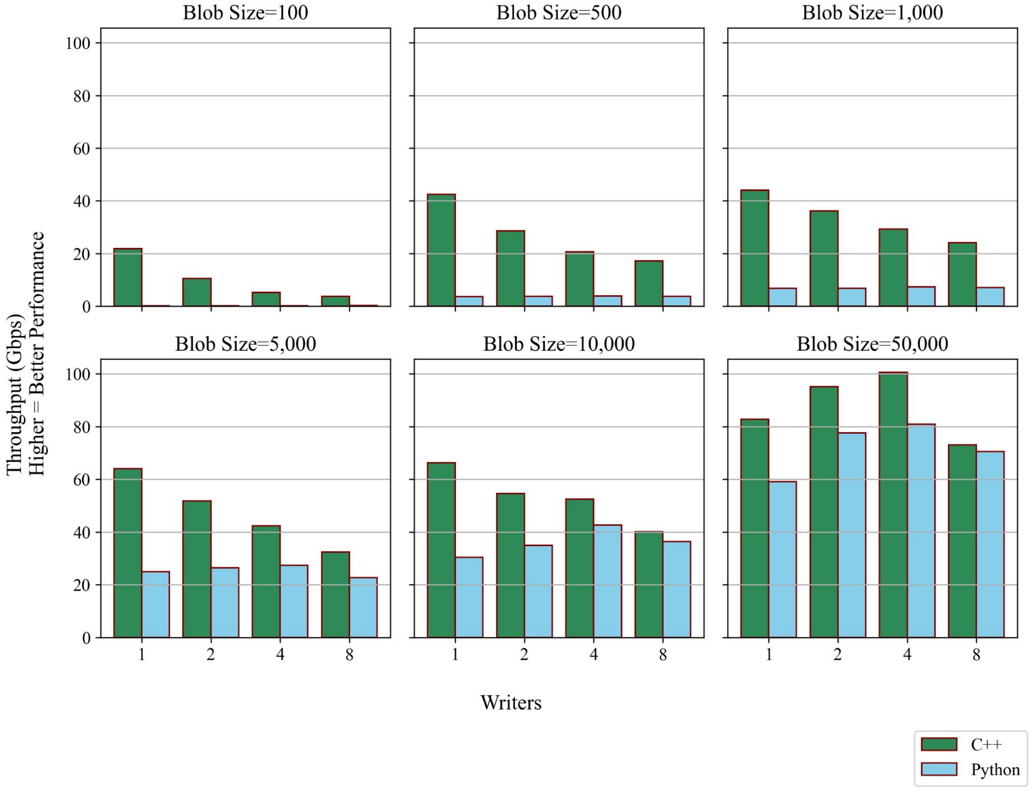
1GB Write Test Results (Latency + Throughput)



1GB Producer-Consumer Test Results (Latency)



1GB Producer-Consumer Test Results (Throughput)



Simple Usage (C++)

```cpp
//Process 1
auto ralA = fral::FRAL("test.bin");
const char *TEST_STR = "TEST";

auto blob = (char *) ralA.allocate(strlen(TEST_STR));
strcpy(blob, TEST_STR);
printf("%s\n", blob);

ralA.append(blob);
```

```cpp
//Process 2
auto ralB = fral::FRAL("test.bin");

while(true){
    auto blob2 = (char *) ralB[0]; //load operation
    if(blob2){
        break;
    }
}

printf("%s\n", blob2);
blob2[0] = 'D';

auto blob3 = (char *) ralB[0]; //load operation
printf("%s\n", blob3);
```

```
TEST
TEST
DEST
```

Simple Usage (Python)

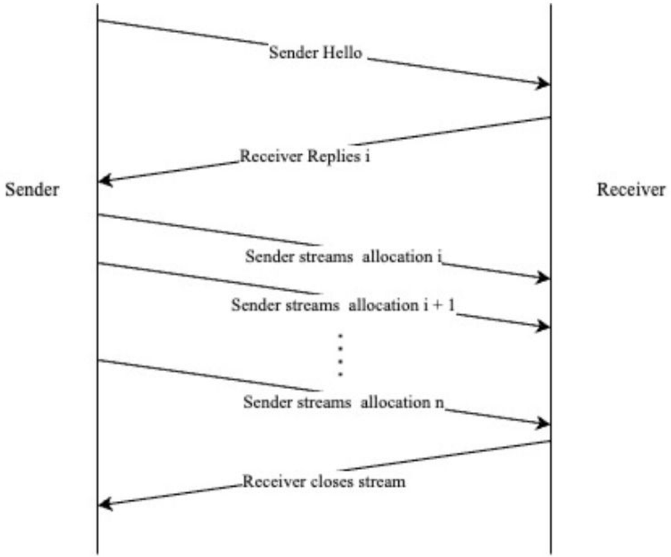
```
# Process 1
ral_A = FRAL("test.bin")
test_bytes = "TEST".encode()

test_blob = ral_A.allocate(len(test_bytes))
test_blob[:len(test_bytes)] = test_bytes

print(bytes(test_blob).decode())
ral_A.append(test_blob)
```

```
# Process 2
ral_B = FRAL("test.bin")

while True:
    test_blob2 = ral_B[0]  # load operation
    if test_blob2:
        break

print(bytes(test_blob2).decode())
test_blob2[0:1] = 'D'.encode()

test_blob3 = ral_B[0]  # load operation
print(bytes(test_blob3).decode()))
```

```
TEST
TEST
DEST
```

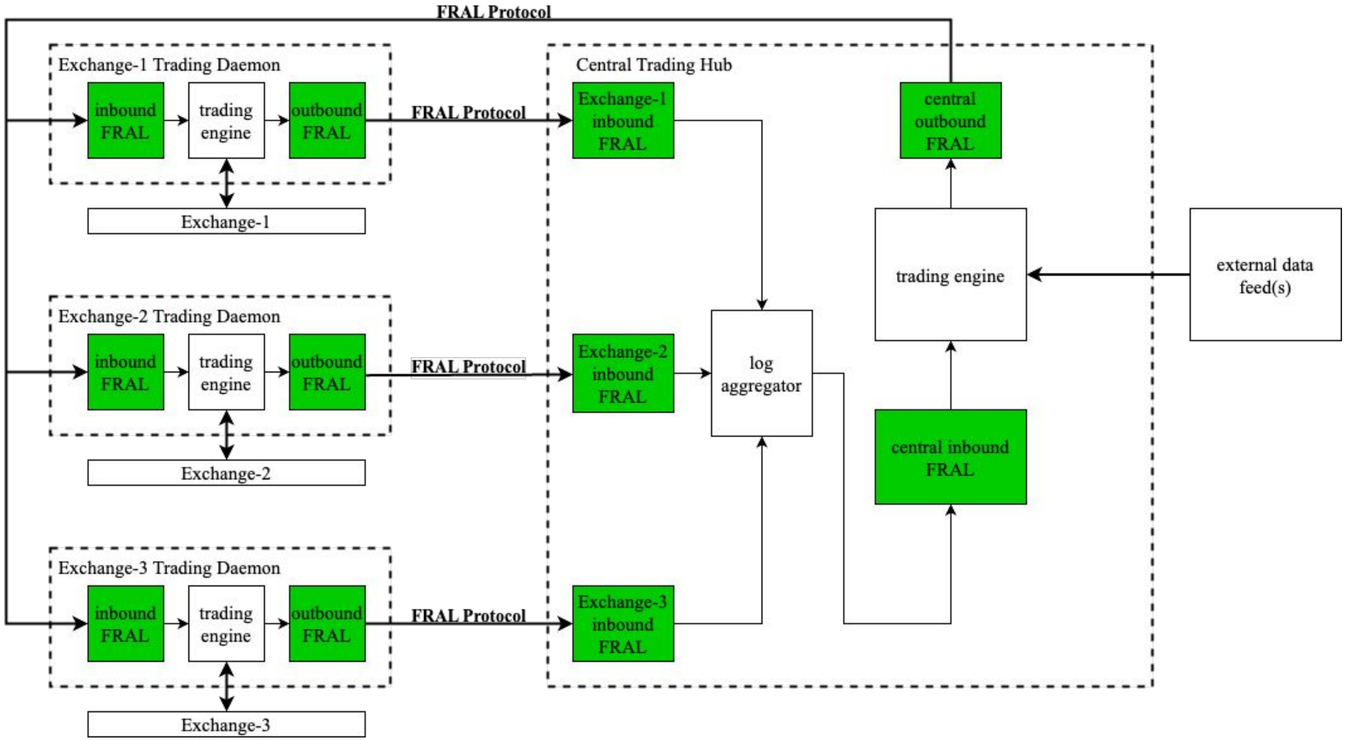
Syncing Over The Network

- Design Considerations:
  - Non-intrusive, i.e., don’t need to alter the underlying log structure or add overhead
  - Order of entries is maintained over network
  - Robust to failure for both client and server; mirroring log is identical even with intermittent failures
- Other details:
  - Initial design Implemented over gRPC and Protobuf
  - Future work: non-intrusive network sync tooling for many-to-one log mappings
- Network Test:
  - Similar to write test, but over the network:
    - Performance degrades (likely to client side buffering by gRPC)
    - Write and Read *N* entries of various sizes

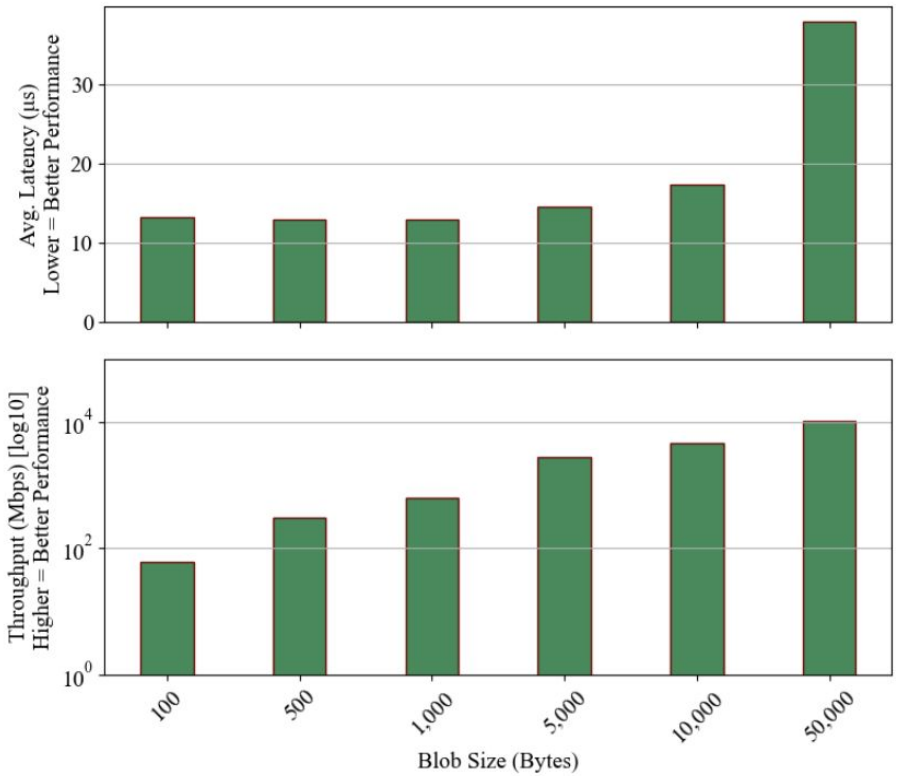


Syncing Over The Network (Use Case)

- *High Frequency Trading:* An entity trades on multiple exchanges in different geographic locations, while needing to keep track of net holdings in a central place



Syncing Over The Network (10k Entry Network Test)



Language-Agnostic Demo

- Language-Agnostic Framework
  - Shared memory opens door for language-agnostic data structures
  - Good for systems with microservices that vary in performance constraints
    - Use C++ for computationally-intensive tasks
    - Capitalize on rapid prototyping in development with Python
    - Example: HFT back-office tasks in Python, trading in C++
  - Demo (video):
    - Read integer from one language, increment by one and write
    - Python initiates process by writing and sending (appending 1)
    - Standard output for writing 5 entries:

```
Python sending 1 to C++!
C++ received 1 from Python, sending 2!
Python received 2 from C++, sending 3!
C++ received 3 from Python, sending 4!
Python received 4 from C++, sending 5!
C++ received 5 from Python, done!
```

Future Work

- Processor affinity functionality
  - Assigning one processor to handle page faults (prefetching)
- Networking performance improvements
- Reliably syncing multiple logs to one over the network
  - Non-intrusive, i.e., don’t add overhead to allocations or underlying log structure as a whole (for accounting)
- High-performance local log aggregator
- Other general performance improvements
  - How does moving away from memory-mapped files affect performance?
- Automatic extension
  - How can we efficiently extend the size of the structure when we run out of space?