

Fast Random Access Log (FRAL)

David Maaghul

Overview

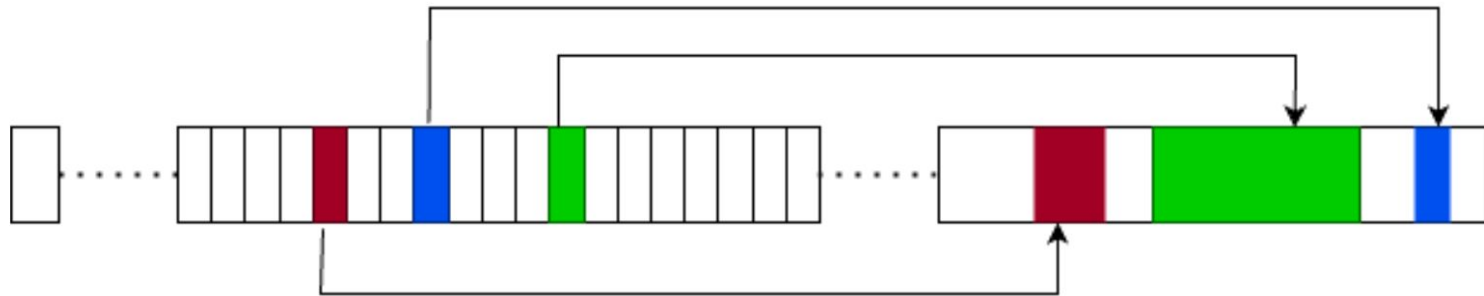
- Goal:
 - Allow multiple processes to read from and write to an array-indexed log as fast as possible
- A Random Access Log:
 - An array-indexed log over shared memory
 - Offers non-blocking writes
 - Reads are random access
- Implementation details:
 - Core engine implemented in C++
 - Python bindings are also provided and tested
- Standard Networking Tools:
 - Implemented and tested tool for syncing logs over network
- Some Use Cases:
 - High Frequency Trading
 - Log Aggregation

Core Architecture

- Three Main Functions:
 - **allocate:** allocates n contiguous bytes of the memory space (a blob) and returns a reference to the beginning of the allocation. This is similar to malloc in C.
 - **append:** appends a memory allocation (blob) to an indexed log, updates the size of the log and returns the index of the appended blob.
 - **load:** returns the memory allocation of a provided log-entry index

Core Architecture (Continued)

- Structuring shared memory:
 - Virtual memory = different addressing for different processes
 - Solution: Store offset from start of shared memory allocation in array
 - Array indexed allocations allows for random access of log entries
 - Array entries are random access
 - Atomic variables:
 - Next available address
 - Next available index (estimation)



Atomic Allocate

- Algorithm uses fetch-add primitive
 - Offset of next free memory address is an atomic variable
- Caveats:
 - Atomic offset variable is unbounded and can grow arbitrarily large
 - Process failure after fetch-add step (line 5) results in wasted space.

Algorithm 1 Atomic 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:    $c = \text{FetchAdd}(h, s)$ 
6:   if  $c + s > S$  then
7:     return null
8:   end if
9:   return  $c$ 
10: end function
```

Atomic Append

- Motivation for approach:
 - What happens if a process fails in the middle of append?
 - Updates atomic index
 - Doesn't index blob
 - Stuck in infinite loop (never break through line 7)
- Want to write an append algorithm that:
 - prioritizes indexing blobs over updating the next-available index variable (atomic)
- Solution:
 - Estimate the next available index

Algorithm 2 Log Reader

```
1:  $S$ : max size of log
2: memspace: the shared memory data structure
3: function READ(memspace)
4:    $i \leftarrow 0$ 
5:   for  $i < S$  do
6:      $blob = \text{load}(\text{memspace}, i)$ 
7:     if  $blob$  is not null then
8:       ProcessBlob( $blob$ )
9:        $i \leftarrow i + 1$ 
10:    end if
11:  end for
12: end function
```

Atomic Append (Continued)

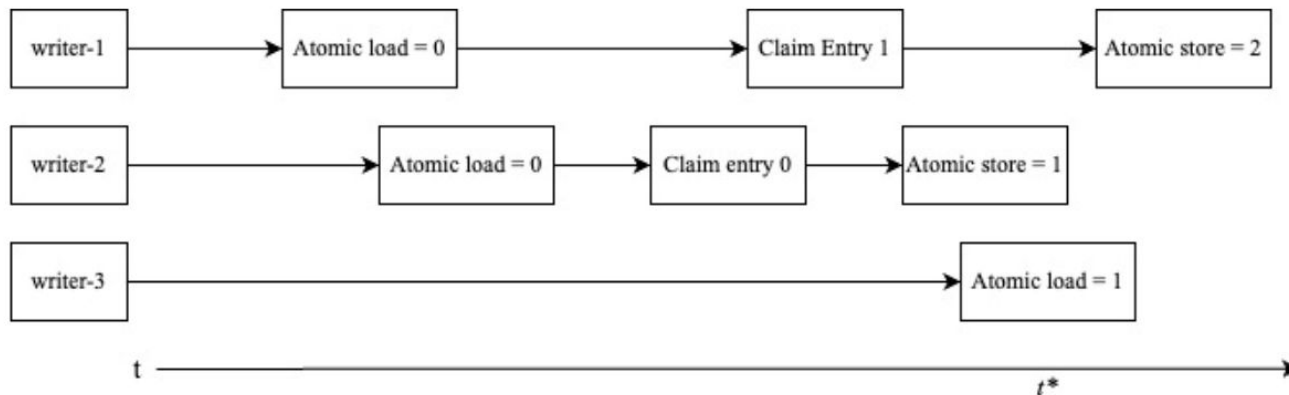
- Use compare-and-swap primitive:
 - Only store offset value if corresponding entry is entry
- A process can fail at any point during the append
 - Array entries are never wasted
 - Iterating over log will never yield infinite loop (previous slide)
- In the case of n writers in parallel:
 - A given writer has a worst case operation count of $O(n)$
 - The total operation count for

Algorithm 3 Atomic Append

```
1: blob: allocation to be appended
2: S: max size of log
3: k: the next available index
4: a: address of start of shared memory space
5: Arr: the log of offsets
6: function APPEND(blob)
7:    $z \leftarrow \text{BlobOffset}(a, \text{blob})$ 
8:    $i \leftarrow \text{AtomicLoad}(k)$ 
9:   for  $i < S$  do
10:    if CompareAndSwap(Arr[i], 0, z) then
11:      AtomicStore(k,  $i + 1$ )
12:      return i
13:    end if
14:     $i \leftarrow i + 1$ 
15:  end for
16:  return -1
17: end function
```

Atomic Append (Continued)

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



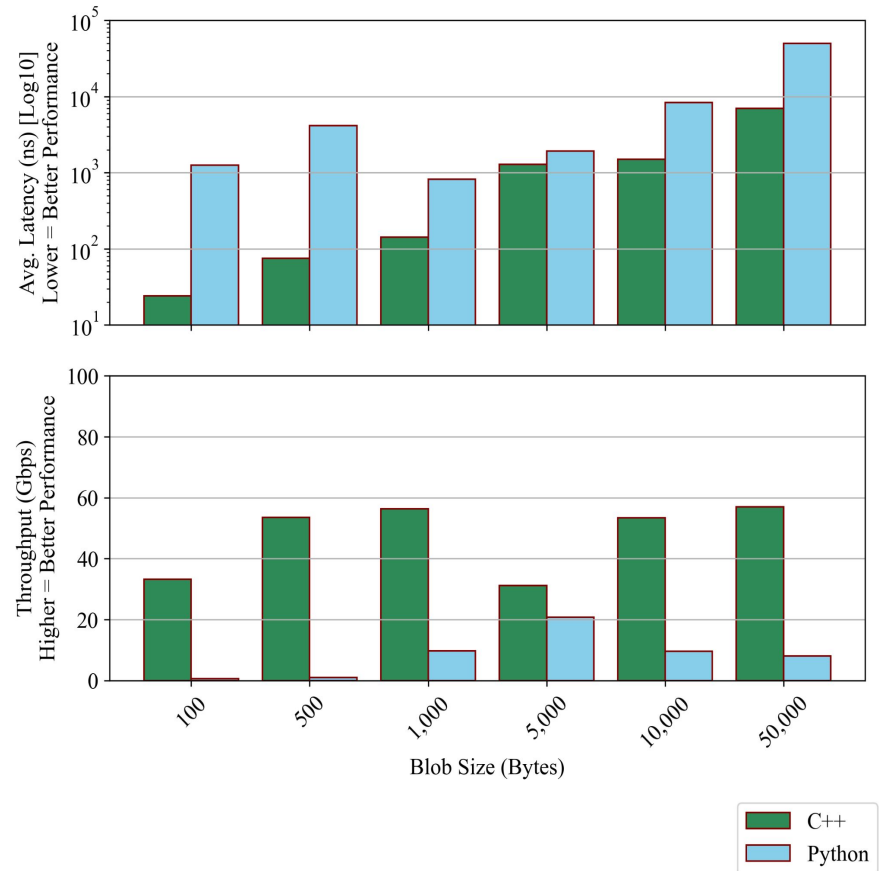
Performance Testing Results

- *Two tests of core architecture performance:*

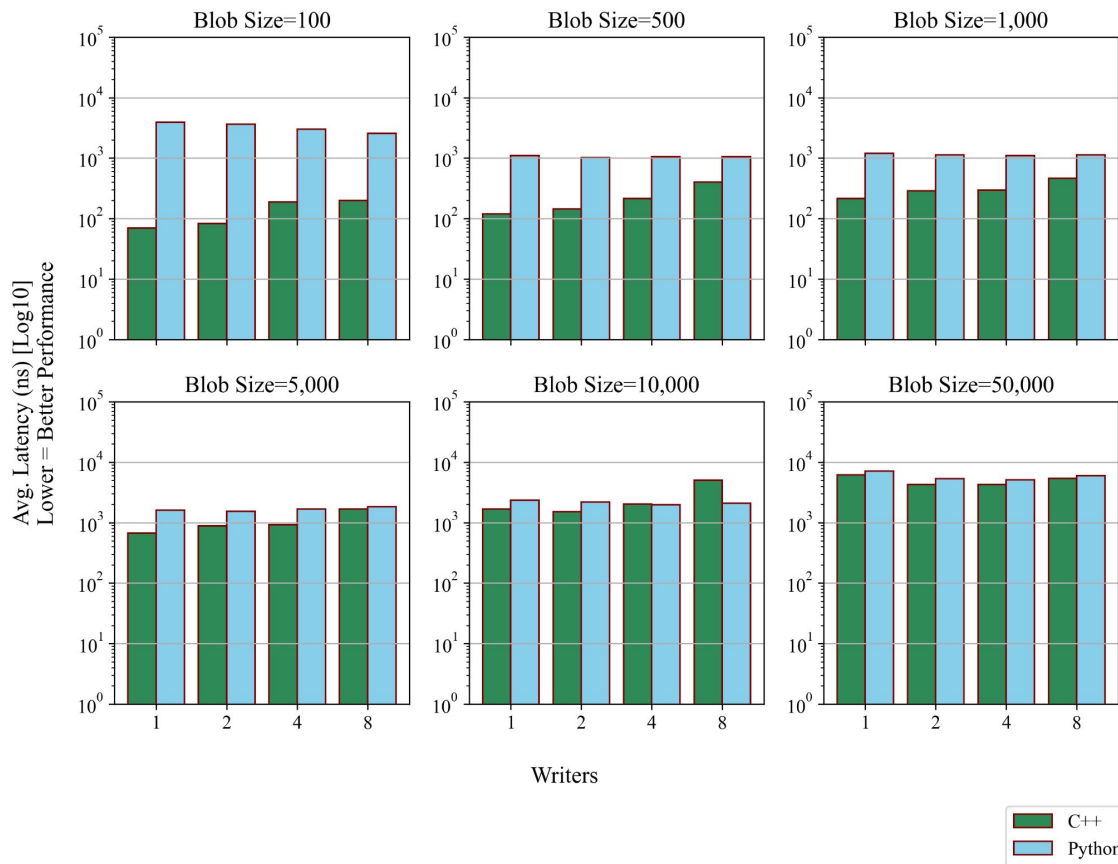
- Look to measure throughput and latency in various ways
- *Write Test:*
 - Measures the speed and efficiency of writing N GB of data to the log in n -byte blobs
 - Assesses performance of high volume of write operations by one writer
 - The timer starts before the first write and ends after the last; 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.

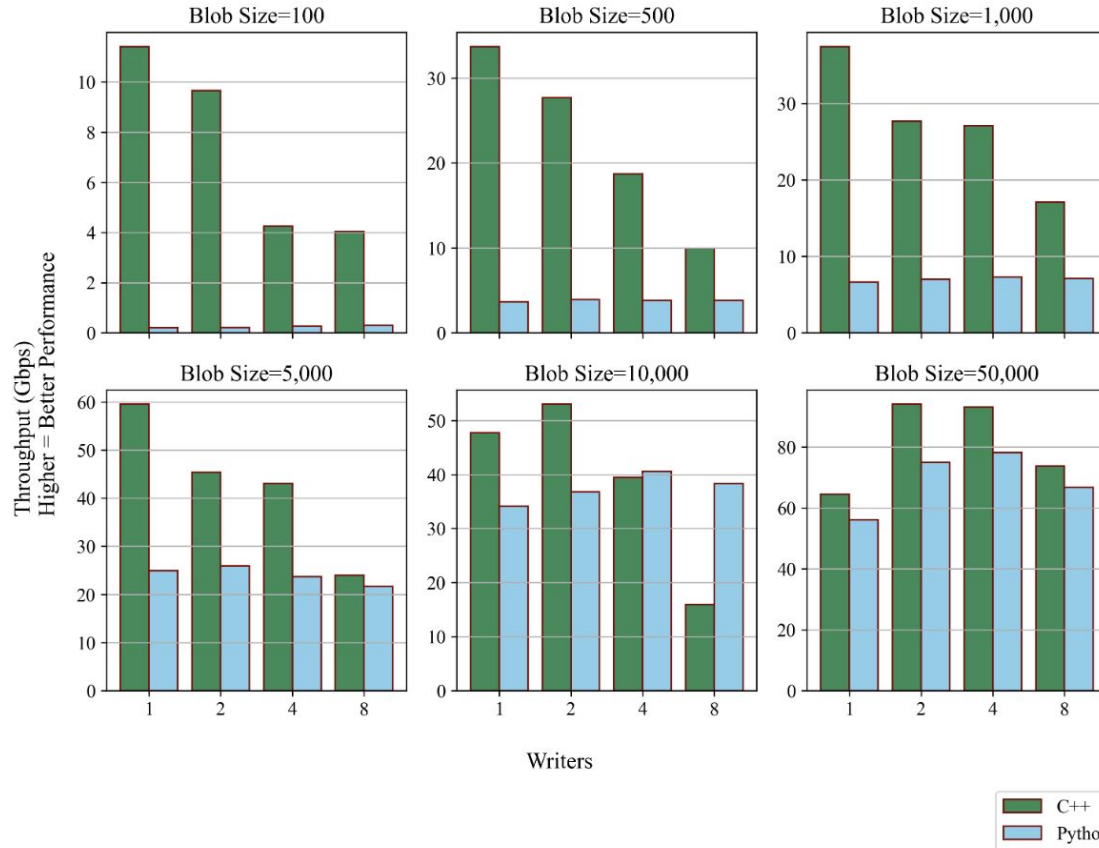
Write Test Results (Latency + Throughput)



Producer-Consumer Test Results (Latency)

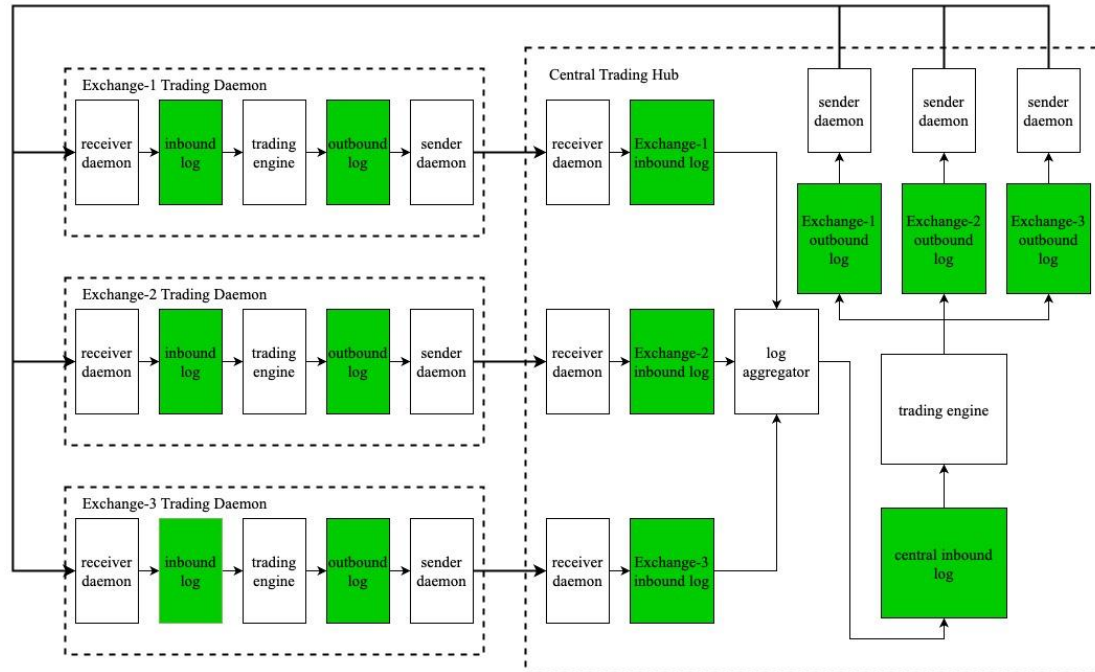


Producer-Consumer Test Results (Throughput)



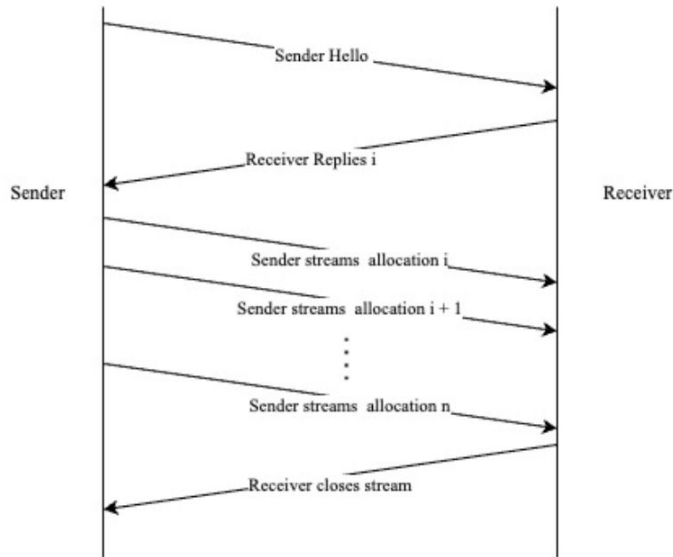
Syncing Over The Network

- A motivating 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

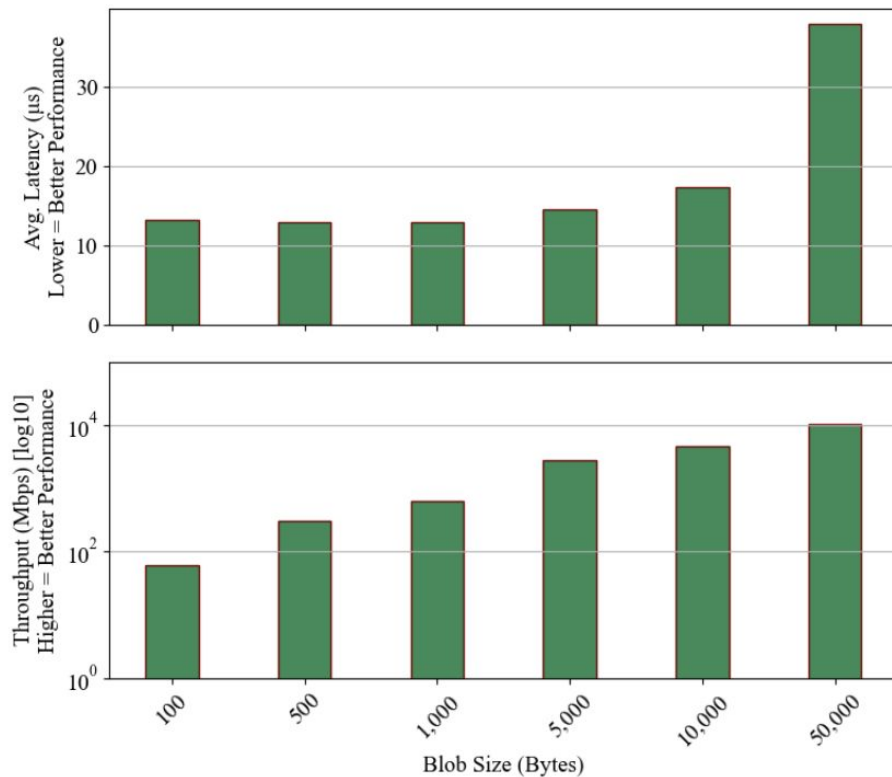


Syncing Over The Network (Continued)

- Intended Requirements:
 - 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, through system failures
- Other details:
 - Initial design Implemented over gRPC and Protobuf
 - Future work: non-intrusive network sync tooling from multiple client logs to map to a single client log
- Network Test:
 - Similar to write test, but over the network:
 - Performance degrades (likely to client side buffering by gRPC)
 - Room for improvement



Syncing Over The Network (Network Test)



Simple Usage (C++)

```
1 //Process 1
2 auto ralA = fral::FRAL("test.bin");
3 const char *TEST_STR = "TEST";
4
5 auto blob = (char *) ralA.allocate(strlen(TEST_STR));
6 strcpy(blob, TEST_STR);
7 printf("%s\n", blob);
8
9 ralA.append(blob);
```

```
1 //Process 2
2 auto ralB = fral::FRAL("test.bin");
3
4 while(true){
5     auto blob2 = (char *) ralB[0]; //load operation
6     if(blob2){
7         break;
8     }
9 }
10
11 printf("%s\n", blob2);
12 blob2[0] = 'D';
13
14 auto blob3 = (char *) ralB[0]; //load operation
15 printf("%s\n", blob3);
```

```
TEST
TEST
DEST
```


Simple Usage (Python)

```
1 # Process 1
2 ral_A = FRAL("test.bin", 1000, 100)
3 test_bytes = "TEST".encode()
4
5 test_blob = ral_A.allocate(len(test_bytes))
6 test_blob[:len(test_bytes)] = test_bytes
7
8 print(bytes(test_blob).decode())
9 ral_A.append(test_blob)
```

```
1 # Process 2
2 ral_B = FRAL("test.bin")
3
4 while True:
5     test_blob2 = ral_B[0] # load operation
6     if test_blob2:
7         break
8
9 print(bytes(test_blob2).decode())
10 test_blob2[0:1] = 'D'.encode()
11
12 test_blob3 = ral_B[0] # load operation
13 print(bytes(test_blob3).decode())
```

```
TEST
TEST
DEST
```

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!
```