IEICE TRANS. INF. & SYST., VOL.E104–D, NO.12 DECEMBER 2021

LETTER

A Case for Low-Latency Communication Layer for Distributed Operating Systems*

Sang-Hoon KIM[†a)], ***Nonmember***

SUMMARY There have been increasing demands for distributed operating systems to better utilize scattered resources over multiple nodes. This paper enlightens the challenges and requirements for the communication layers for distributed operating systems, and makes a case for a versatile, high-performance communication layer over InfiniBand network.

key words: *distributed operating systems, InfiniBand, RPC*

1. Introduction

Recently, many analytic applications require a huge amount of resources to provide the required performance, and the demand is ever-increasing. However, due to the diminishing performance improvement by the end of Moore's Law, it is becoming challenging to provide all the required resources from a single machine. To overcome the limitation of the single machine form factor, there is increasing interest in *distributed operating systems*, where multiple operating system instances (or components) running on multiple nodes collaborate with each other to provide applications with an aggregated view of dispersed resources [1]–[3]. Applications can access abundant system resources without (or with little) modification, thereby transparently improving the application performance, capacity, and overall system resource utilization.

The distributed OS instances should frequently communicate with each other to provide the features of operating systems in a distributed manner. As many OS features such as system calls, file operations, and page fault handling are on the critical path of system performance, the communication layer for distributed OSes should provide low latency as well as high bandwidth. From this standpoint, conventional TCP/IP is ill-suited since its deep stack of layers incurs high communication latencies, and as an alternative InfiniBand has been gaining popularity for building distributed OSes. However, the unique communication model and features of InfiniBand are primarily optimized for high-performance computing (HPC) workloads, and they are not properly aligned to be effectively utilized in the distributed operating system domain. As of the moment of paper writing, there is no standard implementation nor module that can be universally adopted to OS subsystems in spite of the high demands for high-performance in-kernel communication layer. Thus, to realize a distributed OS, existing implementations using the traditional socket abstraction should be rebuilt from the scratch to leverage the high bandwidth and low latency of InfiniBand. The rebuilding is even complicated due to the inherent parallelism and asynchronous nature in operating systems. Thus, this switching demands high engineering effort, and many services resort to IP over InfiniBand (IPoIB) with significant latency of tens of microseconds as opposed to a few microsecond of native InfiniBand. RDMA is one of the most appealing features of InfiniBand. However, it has not been properly discussed in the context and workload of distributed operating systems, and even overly abused without carefully considering the primary roles and goals of operating systems.

This paper makes a case for a versatile, high-performance InfiniBand communication layer for distributed operating systems. While building a distributed operating system, we identified the design considerations that the communication layer of distributed operating systems should take into account. Specifically, we identified a bimodal distribution of network payload sizes and ever-changing locations of target payloads in distributed OSes. These characteristics make the I/O buffer management challenging given that InfiniBand demands constraints for the I/O buffers. In addition, the common design of asynchronous completion processing through a dedicated completion handler can extend the completion path significantly. We enlighten these challenges and requirements, and provide insights to handle them in the prototype implementation.

2. Background

The communication model of InfiniBand is primarily optimized for high-performance computing workloads, and is very different from the traditional synchronous communication model over TCP/IP. Basically, queue pairs accept work requests and their associated completion queues asynchronously deliver the completion of the requests. From the message sender side, it resembles the communication with datagrams in that messages comprising metadata and payload are posted to the underlying layer (a queue pair in this case). The metadata includes the identifier of the request, the type of request, keys to access memory, and so forth. The payload is designated with a scatter-gather list. Before

Manuscript received May 18, 2021.
Manuscript publicized September 6, 2021.
[†]The author is with Ajou University, South Korea.
[*]This work is supported by Electronics and Telecommunications Research Institute (ETRI) grant funded by the Korean government (20ZS1310).
a) E-mail: sanghoonkim@ajou.ac.kr
DOI: 10.1587/transinf.2021EDL8049

the sender posts a message, a receiver should post a receive work request that specifies the address and length of receiving buffer. InfiniBand Host Channel Adaptor (HCA) fills in the buffer with an inbound request and notifies of the event through the associated completion queue. The receiver can inspect the completion queue and processes pending completion events accordingly.

All buffers subject to I/O should be *pinned* to its virtual address space so that corresponding pages are not disturbed by virtual memory mechanisms, such as paging, deduplication, and compaction, during I/O. Also, they should be *mapped* to a DMA-capable memory address to allow HCA to access them directly. In case the buffers are accessed through RDMA, the buffers should be additionally *registered* to HCA as *memory regions*. HCA assigns a remote key for the memory region, and that key is needed to remotely access the memory.

3. Design

3.1 Observation

The communication in *distributed operating systems* exhibits different characteristics compared to that of traditional *user applications*, and we aim to optimize the communication layer considering these unique characteristics. Specifically, one of the common communication patterns in traditional user applications is to distribute a huge amount of data to multiple nodes that process the data in parallel. As we discussed in Sect. 2, the I/O buffers for InfiniBand should be set up properly (i.e., pinned, mapped, and registered) prior to making I/O requests. This setup takes a considerable amount of time, so to amortize the involved overhead user applications usually preallocate huge buffers, set them up during initialization, and use them to the rest of their lifetime. This usage model is possible since the buffers are *solely* and *entirely* dedicated and fixed to the address spaces of the application processes.

In contrast, from the perspective of operating systems, memory is a shared resource, continually allocated to and released from application processes. The source and target data for OS features can be anywhere in the system memory; user applications make system calls with data on their stacks and heaps. Many file and memory accesses are handled from page cache pages that are scattered all over the system memory. Thus, OSes cannot determine which data will be remotely accessed in the future, and this dynamic nature of memory use complicates the buffer management in the communication layer for distributed OSes. This situation is complicated further as some InfiniBand drivers only accept the buffers that are dynamically populated through `kmalloc` or `alloc_pages`. This implies OS cannot simply compose small requests comprising few integers (which are common in distributed OSes as we will discuss later) on its stack, but has to dynamically allocate and release buffers for the messages.

Figure 1 breaks down the time to transfer data over RDMA read in a naïve InfiniBand implementation. We observe that the most of the communication time is spent for buffer management (up to 98.0%), and this portion is so decisive that the actual data transfer time impacts little on the overall performance. This indicates that even though InfiniBand and RDMA are very efficient to transfer data, it might not operate optimally in distributed operating systems spending a considerable amount of time for setting up buffers.

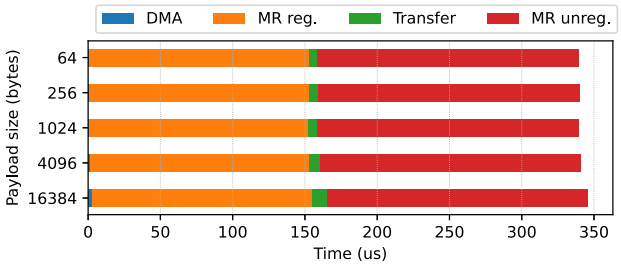

Fig. 1 The breakdown of the time to transfer data over RDMA read. 'DMA' and 'Transfer' imply the time to setup the buffer for DMA and that of actual data transfer, respectively. 'MR reg.' and 'MR unreg.' indicate the time to register and unregister the I/O buffer.

Some studies propose to map the entire physical address space of the system to one big memory region during the system booting time [4]. This approach is, however, impractical in that it can lead to serious security vulnerabilities (i.e, other nodes can access the entire system memory unchecked) and user processes cannot utilize the mapped memory portion for their virtual memory. Worse, as the memory is pinned down to the address space, OS cannot provide virtual memory features from the address space range, which significantly impair the flexibility and efficiency of memory management.

The buffer usage is also different from user applications due to the diverse features that distributed OSes should provide. Generally, we can categorize operating system features into two categories; *controlling the system* and *providing requested data*. For example, process synchronization and signal handling are system control features whereas file and memory operations are data features. The control-related features tend to require small payloads, and their arguments are integer values at most, implying small in size. As they directly control the execution of processes, they are highly latency-sensitive. On the other hand, data-related features involve huge data transfer between nodes. For instance, to provide up-to-date page contents to nodes, pages should be distributed to nodes, and they are bandwidth-sensitive.

Figure 2 summarizes the distribution of message sizes and their contribution to inter-node traffic, which are collected while running a kmeans application on a distributed operating system [3]. We can observe the bimodal distribution of message sizes; small payloads for control-related features and large payloads for data-related features. In terms of message counts, small messages, smaller than a hundred bytes, comprise 74.4% of communication mes-

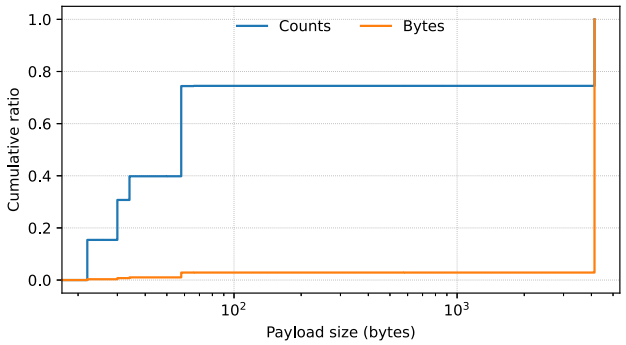

Fig. 2 Analysis on the messages in a distributed OS implementation. 'Counts' indicates the cumulative ratio of the occurrences of the payload size. 'Bytes' indicates the cumulative ratio of transferred data amount. Note that x-axis is in log-scale.

sages. On the other hand, 97.1% of communication traffic comes from big data operations. Therefore, the communication layer should be designed keeping these characteristics in mind.

Lastly, operating system internals are inherently concurrent and asynchronous. Multiple processes can simultaneously invoke system calls, and multiple cores may be handling interrupts simultaneously. Moreover, it is common in operating systems to process operations asynchronously (especially for I/O) to increase overall processing throughput. Thus, the communication layer should provide a proper communication model that fits these sophisticated features and implementation in distributed OSes.

3.2 Design and Implementation

Message Sending. Based on the observations, we concluded that efficient buffer management is the most crucial part for the communication layer in distributed OSes. To amortize the high buffer setup overhead, we opt to preallocate a small memory region during initialization and to use it as a *send buffer* for communication. Specifically, during the distributed OS startup, each node establishes connections to all other nodes specified in a configuration. For each connection, a small amount of memory (configurable and 32 MB by default) is allocated, registered as a memory region, and associated with the connection. A sender (i.e., a component in the distributed OS) can allocate a send buffer from the memory region and compose a message on the buffer directly. Also, we maintain a pool of send work requests to facilitate asynchronous message processing. In posting the message, the sender may choose to be synchronous (wait until the message is processed) or asynchronous (proceed without waiting for the completion). The buffer and send work requests are automatically reclaimed after the posted message is processed.

To support concurrent and parallel communication, multiple senders can simultaneously allocate multiple send buffers from the preallocated memory region. Under the hood, the memory region is populated with multiple, physically contiguous 4 MB chunks (in the Linux kernel we can allocate physically contiguous pages up to that size from the buddy system allocator), and the chunks are logically organized into a circular buffer. Buffer allocation requests are served from the head of the circular buffer unless it touches the tail of the buffer. When a buffer is reclaimed, its space is marked as free, and the tail of the circular buffer is updated accordingly if the reclaimed buffer is at the tail. If a buffer is not returned quickly, it can straggle the tail of the ring buffer, making the ring buffer full. In this case, buffers are temporarily allocated with `kmalloc` until the straggling buffer is released.

Usually, communications especially for system control purposes should be synchronously handled. This implies the communication layer should notify senders of the message processing completion fast. However, we can only attach one send completion queue for one connection, thereby enforcing to demultiplex completion events to multiple senders. In our previous work, we implemented the completion notification mechanism that senders are blocked after posting messages and a dedicated thread pumping completion events out of the completion queue wakes up blocked senders. This approach incurs too much latency (up to hundreds microseconds) due to the context switch and inter-thread communication overhead. We optimized the completion path by removing the dedicated thread and making senders keep polling the completion queue after posting a message. Specifically, each sender has an associated flag and its address is embedded in the send work request. When a sender gets a completion event from the completion queue, it sets the flag specified in the completion event. After setting the flag, the sender checks its own flag and may proceed once the flag is set. Otherwise (i.e., the flag was for other sender), the sender polls the completion queue again. This way, we can significantly reduce the notification latency when a single thread is accessing the connection which is the common case.

The proposed approach can involve one memory copy while loading the payload. Many control messages are, however, very small in distributed OSes, and the memory copy overhead is much smaller compared to the buffer setup overhead. Nevertheless, the sender may specify the address of payload instead of using the preallocated buffer in case it want to send large buffers without memory copy and/or over RDMA.

Message Receiving. Likewise the send buffer, we initialize the system with a number of receive work requests posted. The receive buffers for the work requests are preallocated along with the receive work requests. When a sender posts a send work request, its payload is transferred to the receiving node and filled in one of the receive buffers. And the receiving event is notified through the completion queue mechanism.

We implement an interface that resembles that of traditional RPC systems. An OS component can register a message type and its associated handler to the communication layer. When the system receives a request, it checks the type of the received message and forwards it to its registered han-

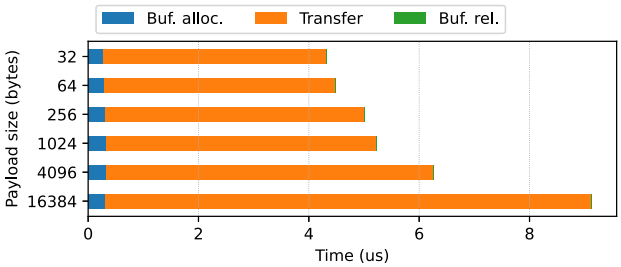

Fig. 3 The breakdown of the time to send messages over proposed system. 'Buf. alloc.' and 'Buf. rel.' mean the time to allocate and release I/O buffer using the proposed scheme. 'Transfer' implies the time taken to transfer data.

dler. To facilitate parallel message processing, we maintain a pool of worker threads, and dispatch received messages to the workers. Thus, the handler is executed in the context of the worker threads. Upon returning from the handler, the received message and associated receive work request is automatically recycled for subsequent requests.

By distributing inbound requests to a pool of worker threads, the requests are processed in out-of-order. However, we found that there are some cases in a distributed OS where multiple messages should be handled in order. To this end, OS components can specify while registering the handler whether the message type should be processed in an order or not. The messages of these types are dispatched to a designated worker and processed in a serialized manner in the context of the worker.

4. Evaluation

We evaluated the proposed communication layer with four nodes each of which is with one Intel Xeon Gold 5215 processor and 128 GB of memory. The nodes are equipped with Mellanox ConnectX-5 InfiniBand HCA, and they are connected with a Mellanox SX6012 InfiniBand switch which provides 56 Gbps bandwidth.

Figure 3 summarizes the performance of the communication layer. We measure the time for each communication step in the implementation. 'Buf. alloc.' and 'Buf. rel.' mean the time to allocate and release I/O buffers as mentioned in Sect. 3.2, and they only account for 3.5% to 6.6% of entire message sending time. This indicates that our implementation significantly reduces the overhead for buffer management compared to the time spent setting up buffers in Fig. 1.

To evaluate the performance on multithreaded case, we measure the operation processing rate while changing the number of sending threads and message sizes. Figure 4 shows the aggregated operations per second on various message sizes and threads. We can observe the aggregated ops increases as the number of threads is increased on all message sizes. From this result we can confirm the scalable performance of the proposing communication layer.

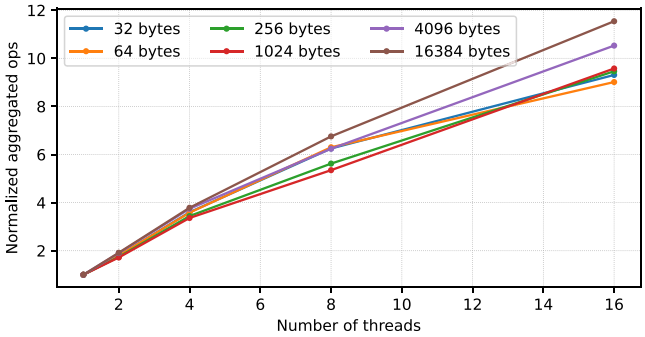

Fig. 4 The trends of aggregated operations per second on various message sizes and sending threads. The ops is normalized to the single thread performace.

Using the proposed communication layer, we implemented a distributed operating system [3]. The distributed OS is based on Linux and allows a process to be run split across multiple nodes. To this ends, the system defines 18 message types for providing various OS features such as process migration, memory consistency protocol, process synchronization through futex, and signal handling. We believe this demonstrates the versatility of the proposing communication layer.

5. Conclusion

In this paper, we presented the unique characteristics of the communication layer for distributed operating systems, and provides optimization techniques tailored to the characteristics. We implemented the proposed scheme in a real distributed operating system, and verified that the proposing scheme eliminates the most of the buffer management overhead. We are working on implemeting additional features at the communication layer, including payload replication over multicasting and commit-based checkpointing, to simplify communication patterns that are common in distributed OSes.

References

[1] Y. Shan, Y. Huang, Y. Chen, and Y. Zhang, "LegoOS: A disseminated, distributed OS for hardware resource disaggregation," Proceedings of the 13rd USENIX Symposium on Operating Systems Design and Implementation (OSDI), Carlsbad, CA, Oct. 2018.

[2] J. Zhang, Z. Ding, Y. Chen, X. Jia, B. Yu, Z. Qi, and H. Guan, "GiantVM: a type-II hypervisor implementing many-to-one virtualization," Proceedings of the 16th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE), pp.30–44, March 2020.

[3] S.-H. Kim, H.-R. Chuang, R. Lyerly, P. Olivier, C. Min, and B. Ravindran, "DEX: scaling applications beyond machine boundaries," Proceedings of the 40st International Conference on Distributed Computing Systems (ICDCS), 2020.

[4] S.-Y. Tsai and Y. Zhang, "LITE kernel RDMA support for datacenter applications," Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP), Shanghai, China, pp.306–324, Oct. 2017.