Light-Weight Remote Communication for High-Performance Cloud Networks

Jens Kehne, Marius Hillenbrand, Jan Stoess, Frank Bellosa

System Architecture Group, Karlsruhe Institute of Technology, Germany
{kehne, hillenbrand, stoess, bellosa}@kit.edu

***Abstract*—In this paper, we present early experiences with *libRIPC*, a light-weight communication library for high-performance cloud networks. Coming cloud networks are expected to be tightly interconnected and to show capabilities formerly reserved to high-performance computing. LibRIPC aims to bring the benefits of such architectures to heterogeneous cloud workloads.**

LibRIPC was designed for low footprint and easy integration; it supports reconfiguration and mutually untrusted communication partners. LibRIPC offers short and long transmit primitives, which are optimized for control messages and bulk data transfer respectively. Early experiments with a Java-based web server indicate that libRIPC integrates well into typical cloud workloads and brings substantial speedup of at least a factor of three for larger data transfers compared to socket-based TCP/IP communication.

I. Introduction

Over the last decade, the Internet architecture has converged from a paradigm of loosely coupled distributed nodes towards a centralized structure, with data centers and cloud platforms, typically consisting of tightly interconnected commodity computers hosted in one large facility, being the principal agglomerations. Most parts of the hardware were designed specifically to be operated in a massive data center [1], [2]. Furthermore, coming cloud and data center processors will not be able to sustain the per-core performance of previous, less scalable systems, and large numbers of low-power chips such as ARM or Intel Atom are expected to become prevailing technology.

These trends cause an orientation shift from pure "compute power" towards a balanced system design, where power and capabilities of the interconnects are as crucial as those of the processors. Modern cloud interconnects increasingly focus on high bandwidth and low latency on the one hand, and balanced design and power awareness on the other: HPC network fabrics such as InfiniBand or Blue Gene are being investigated in cloud computing setups [3]–[6]; their underlying technologies such as user-level I/O and remote DMA are gaining momentum also for commodity Ethernet networks [7]. Research is also investigating power-efficient network alternatives such as PCIe for their suitability as inter-node fabric [8].

Cloud workloads are immensely complex, however, and interfacing cloud applications with increasingly powerful network fabrics has become a challenging problem. Workload types range from conventional multi-tiered architectures (Web Services), over memory-intensive distributed systems (Object Caching, Stream Computing), to large-scale data processing platforms (Social Messaging, Big Data). Often, they comprise a heterogeneous variety of smaller, already complex components, and they make use of a wide variety of programming paradigms, systems and runtime environments, encompassing everything from low-level interfaces through virtual machines to managed runtime systems such as Java and Python.

There has been an increasing amount of efforts that investigate how efficient communication can be facilitated across heterogeneous cloud components. To the best of our knowledge, none of these approaches have specifically explored how coming interconnect architectures and their anticipated high-performance capabilities can be leveraged for efficient, message-based communication in the cloud in a generic and easy-to-integrate fashion.

In this paper, we present early experiences with *libRIPC*, a light-weight communication library for high-performance cloud networks. LibRIPC targets heterogeneous cloud workloads with multiple, potentially untrusted domains, and strives to deliver high performance as well as flexibility and ease of integration. Instead of the traditional socket-based networking semantics, libRIPC provides a message-based communication interface atop a flat endpoint space, allowing it to better exploit the tight coupling and the advanced networking capabilities (e.g., user-level I/O, remote DMA) of present and coming high-performance cloud networks. LibRIPC offers two main functions, short and long send, separating the signaling path from the bulk data transport path. In contrast to HPC communication stacks like MPI, libRIPC does not assume mutual trust between communication partners; libRIPC primitives perform effectively even when endpoints do not trust each other. Early experiences with integrating our library into a Java-based web server indicate that libRIPC integrates well into typical cloud workloads and brings a substantial speedup of at least a factor of three for larger data transfers. The remainder of the paper is structured as follows: § II presents a motivation for message-based communication, while § III presents libRIPC's basic architecture. § IV presents our initial evaluation, followed by related work in § V and a summary in § VI.

II. The Case for Message-based Communication

In the traditional model of server computing and networked applications, companies posses and operate their own computers, which are based on commodity components and operating

systems. The protocols and abstractions used in these operating systems are still largely the same that enabled the rise of the Internet in the 1980s, the most important ones being the Berkeley socket interface and the TCP, UDP, and IP protocol suites. They are still well-suited for today's Internet communication: TCP/IP, for example, enables efficient and reliable communication over unreliable, wide-area networks. Berkeley sockets, conversely, provide an easy-to-use interface which maps well to the semantics of TCP and UDP.

Since the last decade, companies have increasingly moved applications from their own, dedicated hardware to external data centers in order to save costs of ownership and operation. The data centers, in turn, are thus faced with the demand to host and support a variety of different applications; as a result, they resort to the same commodity hardware and operating systems their customers' applications were designed for. Consequently, the internal networks of current data centers rely heavily on TCP/UDP/IP and Berkeley sockets in order to maintain application compatibility. These internal networks, however, are fundamentally different from the Internet that TCP/UDP/IP were designed for: Endpoints are physically close to each other, resulting in low interconnect latencies, low failure rates and no need for routing or fragmentation in software, rendering TCP/UDP/IP's most important functionalities largely superfluous.

In addition, we expect special features currently found mostly in high performance computing – like protocol offloading, user-level I/O and remote DMA – to become increasingly common in future datacenters. Some of these features, such as remote DMA, imply message-based semantics: The hardware is programmed with a network transaction – such as sending a packet – and then executes that transaction autonomously. On the other hand, many applications use high-level protocols based on self-contained messages and thus could, in principle, exploit such special features. For example, both HTTP requests sent to a web server and the files returned in response to these requests can be regarded as self-contained messages. However, the socket interface does not preserve information about the boundaries between messages, forcing messages to be fragmented. In addition, since a message can often not be processed until it has been fully received, a receiving application can be forced to poll its socket repeatedly until a complete message has been received. Such applications would be better served by a message-based communication interface.

Since interconnect hardware is often too complex to be used directly by applications, an abstraction layer is needed in order to facilitate application development. The interface offered by that abstraction layer to applications should be simple enough to allow for easy application development, yet close enough to the native hardware semantics to achieve high performance. Fortunately, high-performance interconnects typically share the same core functionality and differ mainly in the details. By exploiting the common features found in all interconnects, the interface we envision offers a simple messaging interface to the application, while hardware-specific details are handled transparently by an interface library.

III. LibRIPC Design

The design of *libRIPC* is driven by the characteristics of present and future, tightly-networked cloud applications [1]. We anticipate that, in the near future, such environments will feature capabilities formerly reserved to HPC architectures, including user-level I/O, protocol offloading, and remote DMA (rDMA). We also expect such environments to increasingly trade hardware reliability for power, cost, and scale, leading to increased rates of failure and imbalance. Software-wise, we assume heterogeneous, federated workloads showing different communication patterns including request/reply and peer-to-peer schemes as well as distributed signaling and object access, and different, from very small to very large, transfer volumes. We derive the following design goals:

Low footprint: Many coming cloud workloads will have strong demands on network latency and throughput. Our library should enable low-overhead communication, avoid copying overhead, and make use of high-performance hardware whenever possible.

Ease of integration: Our library should integrate easily into the wide variety of cloud applications. Given high-performance networking typically involves user-level I/O, the library should interface with different run-time environments and allow generic access to different high-performance network hardware.

Support for reconfiguration: Nodes, networks, and applications may fail or be reconfigured (join, leave) at run time, and thus require a high degree of flexibility and fault-tolerance. Our library should treat reconfiguration as common case, with endpoints migrating or even disappearing during normal operation.

Safety for untrusted partners: In contrast to cooperative workloads like HPC, communication partners are not necessarily inside the same trust domain. We thus need to support high-performance communication among untrusted parties.

A. LibRIPC Communication Primitives

LibRIPC differs from conventional network layers such as TCP/IP in that it assumes very close interconnection; communication is message-based rather than stream-oriented, and libRIPC does not provide any means for fragmentation or flow-control. LibRIPC resembles the minimalistic design of in-kernel high-performance messaging systems like in L4 [9], extending it to remote communication. LibRIPC strives to achieve performance comparable to HPC communication libraries such as MPI, but for heterogeneous workloads with varying communication and trust patterns. Messages are always transferred in their entirety, leading to lower waiting times at the receiving end, as there is no need to wait for more data on a socket. In order to separate the path for signaling from the path for bulk data transport, libRIPC offers two main functions, one for short and one for long messages (Fig. 1).

LibRIPC assumes direct access to the network hardware from applications and avoids copying data as much as possible. Send data is always passed as pointers and read directly from its location in memory. The receive path places data directly in

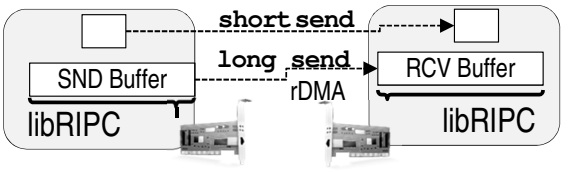
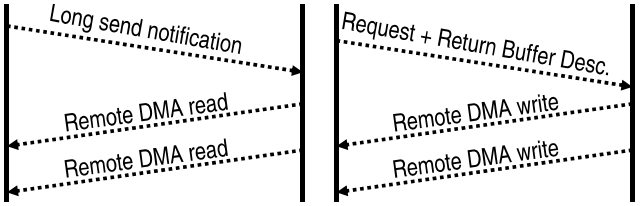


Figure 1. LibRIPC design.



(a) Without return buffers (b) Using return buffers

Figure 2. Long send without and with return buffers.

the user's address space, returning a pointer when the transfer is finished. To ease porting across different network architectures, libRIPC's interface is hardware-independent. LibRIPC addresses services (i.e., programs) rather than physical machines, allowing services to join or leave the network, or to migrate between machines without disturbing communication.

Short Messages

Short messages are optimized for signaling purposes such as synchronization, remote interrupts, or request distribution. Such messages are usually small, but tend to have tight latency requirements. For example, read requests to a database are often small, but their latency should be low to minimize waiting for the response. Short messages require no negotiation between the sender and receiver, which lowers the overall latency. The amount of data transmitted is limited by the receive buffer size and possibly other factors (e.g., the MTU). Sending messages larger than the limit will fail as messages are not automatically fragmented. Some hardware platforms have dedicated support for interrupt and signaling across nodes (e.g., barrier network on Blue Gene/P). If available, libRIPC exports these capabilities through the short message protocol.

Short messages contain the source and destination service IDs, the number of data items, and an array of individual payload items. If possible, libRIPC uses user-level I/O to send the message payload directly from the sender to the receiver address space. To further lower latency, libRIPC short send does not implement reliability, dropping messages silently on transmission failures. Users are free to implement their own reliable protocol atop.

Long Messages

Long messages allow carrying larger payloads. For long send, libRIPC assumes rDMA capabilities to directly store or read data from a remote node. Since sender and receiver may not trust each other, and endpoints may service huge numbers of clients not known in advance, it is not feasible to allocate static memory segments for rDMA. Unless the rDMA engine features append semantics (which current hardware does not), it is also not possible to provide a safe, shared receive buffer for multiple clients [10].

LibRIPC long send therefore resorts to a two-phase protocol comprising a short control message and a follow-up rDMA read (Fig. 2(a)): the sender puts the payload into a buffer accessible by the rDMA engine. The client then sends a (short) control message to the recipient, which describes one or more data buffers on the sender side. The recipient then performs a rDMA read operation for each buffer to copy the data from the sender's memory directly into a receive buffer. This protocol allows a server to place a large response in its own memory and then serve the next request, leaving the transfer to the client. As the recipient is free to specify the location and size of the receive buffers, the size of long messages is principally unlimited. Note that, since the recipient's rDMA reads are invisible to the sender on typical network architectures, the recipient notifies the sender after the complete read cycle in order to allow releasing send buffers. This notification, however, is not part of libRIPC long send, as it could be issued explicitly, piggybacked to follow-up messages, or even completely omitted in favor of a timeout protocol.

Return buffers

As alternative to the default long send path, libRIPC introduces the concept of *return buffers*. It is inspired by the works in [11] and is based on rDMA writes instead of reads: here, the sender allocates a number of return buffers, and then attaches buffer descriptors to a short message (Fig. 2). The recipient may later issue direct rDMA writes into those return buffers. While the number of round trips required with and without return buffers stays the same, return buffers allow choosing the particular communication partner where the transmit phase should run. A heavily loaded client can resume other work while the server is writing the reply into the client's memory. When the client finally calls the receive function, the transfer is already complete and computation can resume immediately. Conversely, a heavily loaded server may choose to ignore the return buffers and revert to the regular long send protocol.

Addressing

LibRIPC uses hardware- and location-independent addresses, which we call `service IDs`, to name endpoints. Applications register one or more service IDs with libRIPC before sending or receiving messages. Service IDs are resolved to physical addresses dynamically and transparently to the application. Applications can also migrate between machines or crash and restart with the same service ID. In both cases, libRIPC will transparently resolve the new location of the process and communication will resume seamlessly.

Our current implementation uses a multicast-based resolver which works similar to the ARP protocol for Ethernet and IP, which has the advantage that services can find each other without needing to know the location of a central directory beforehand. Another solution would be to use a (possibly distributed) directory service that holds information about

all services on the network, and use our current multicast scheme to initially discover the location of the directory. After discovery, services could query location information from the directory service instead of broadcasting. A central service would also ease detection of service migrations, as migrated services could explicitly notify the directory of the change.

IV. Initial Evaluation

As initial application scenario, we have integrated libRIPC into the Jetty HTTP server [12]. Jetty is a full web server with support for dynamic content, for stand-alone or embedded use. We chose Jetty because it is used in several well-known software projects, including Apache Hadoop [13], Eclipse and Google AppEngine. Jetty allowed us to explore how libRIPC, which is written in C, can be used efficiently in other runtime environments such as Java. Jetty interfaces with libRIPC through Java Native Access (JNA) [14].

For data management, Jetty's core uses *ByteBuffers* managed by the JVM. JNA supports direct access to memory allocated by C code through the same ByteBuffer class, allowing us to pass data between Jetty and libRIPC without copying. In order to emulate the synchronous behavior Jetty expects, our implementation transparently swaps buffers containing sent data with new, empty ones, giving Jetty the illusion that buffers are empty after transmit. Similarly, when receiving data, we swap the empty buffer meant to contain the received data against a receive buffer returned by libRIPC's receive function.

We have initially evaluated libRIPC on an InfiniBand (IB) cluster, with nodes equipped with 2.26 GHz Xeon E5520 CPUs and 6 GiB RAM each, and the network comprising Mellanox ConnectX-2 adapters connected through a DDR (16 GBit/s) switch. We used CentOS 5 with 64-bit Linux kernel 2.6.35. We considered performance and ease of integration as the most interesting evaluation criteria. Integration of libRIPC into Jetty was surprisingly simple. We finished a first working implementation within a week; the current, improved version amounts to about 400 lines of Java code. No modifications to the JVM were necessary, and the implementation does not assume any details about the network hardware.

To evaluate performance, we compared Jetty/libRIPC to the vanilla, socket-based version. The benchmark lives on a remote node and issues HTTP GET requests to Jetty, requesting files filled with random data of increasing size. For fair comparison, we ran the vanilla Jetty on the InfiniBand network as well, through the IP-over-InfiniBand emulation layer of the Linux kernel. For each file size we tested, we fetched the same file ten thousand times, using the same files for both versions of Jetty. Fig. 3 shows the results. Vanilla Jetty features an internal cache that keeps content buffers smaller than 10 MiB (by default) in RAM after sending, indexed by a hash table. Our modified Jetty sends those cached buffers directly via long send if present. With this cache enabled, libRIPC always outperforms the vanilla implementation, for larger file sizes by a factor of three. With the cache disabled, however, libRIPC performs worse than the socket implementation for file sizes smaller than about 6 MiB. This is owed to differences in the file read

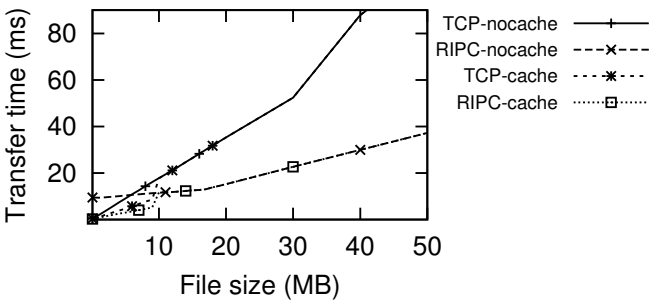


Figure 3. Performance of Jetty file fetch over libRIPC compared to sockets and IP-over-InfiniBand. Note that files larger than 10 MB are not cached.

paths: vanilla Jetty uses `mmap` to load files into memory, while Jetty/libRIPC presently uses slower `sys_read`s to fetch data into rDMA buffers. We are working on an implementation that allows rDMA on `mmap`ped files as well.

Altogether, we draw the initial conclusion that libRIPC poses a promising approach to deliver high performance communication to today's and coming cloud workloads. However, we expect the results to vary with the features present in the underlying network hardware: On BlueGene/P, the long send may perform better since no memory registration is required, while on iWARP, the short send may perform worse as iWARP supports user-level I/O only for remote DMA. Also, other applications may not integrate as well with libRIPC as Jetty. Since such effects are difficult to quantify, further exploration on different network hardware as well as other application scenarios is necessary.

V. Related Work

In presence of the growing complexity and heterogeneity of cloud workloads and runtime environments, there has been an increasing amount of efforts investigating efficient communication across heterogeneous cloud components: Enterprise Service Busses attempt to ease interconnecting SOA components; messaging middlewares such as ZeroMQ [15] aim to ease developing scalable distributed applications independent of the programming language. To the best of our knowledge, none of these approaches have specifically explored how coming interconnect architectures and their anticipated high-performance capabilities can be exploited and integrated for efficient, low-latency communication in the cloud.

The Sockets Direct Protocol [16] accelerates standard socket semantics by incorporating low-level network link capabilities such as remote DMA. Applications communicate directly using zero-copy, OS-bypass transfers from within their address space, without relying on a TCP/IP stack. MegaPipe [17] eliminates most of the overhead associated with sockets and extends their functionality by asynchronous I/O. The most notable difference of both to libRIPC is that they still mimic socket semantics, while our library strives to move away from sockets in favor of more elementary primitives. Wang and colleagues have extended Hadoop to take advantage of InfiniBand [3]. Similarly in spirit, works by Jose et al [4]

and by the authors themselves [5] have investigated how a distributed memory cache can exploit HPC interconnects such as InfiniBand or those of the Blue Gene/P supercomputer for better efficiency. LibRIPC strives to generalize such efforts both with respect to the cloud workloads running atop and the hardware architectures running underneath.

The Message Passing Interface [18] is the dominant tool for communication in high-performance computing. Like libRIPC, it offers a message-based interface and transfers messages atomically, achieving both high performance and low latency. However, MPI, comprising over 300 functions, is too complex for use in general-purpose applications. MPI also assumes mutual trust between all communicating nodes, provides only limited fault-tolerance, and does not support restarting or migrating of processes. We therefore conclude that MPI lacks the flexibility necessary for cloud applications.

The Common Communication Interface [19] is similar to our approach in that it provides a simple and hardware-independent interface to high-performance hardware. It offers message-based, zero-copy communication and leverages hardware features to achieve high performance. However, CCI relies on connections between communicating endpoints, which incur management overhead. CCI also exposes the exchange of pointers and subsequent remote DMA transactions to the application. While this may improve performance for certain applications, we believe that simple send/receive semantics are more desirable for others.

Finally, our approach shares the message-oriented interface and the flat endpoint identifier space with the Fast Local Internet Protocol (FLIP) [20] of the Amoeba distributed system [21]. Also related are the send and receive window semantics of the IPC primitive of the L4 micro-kernel [9]. Effectively, our approach strives to extend those approaches to high-performance interconnects and remote communication.

VI. Conclusion

In this paper, we have presented *libRIPC*, a light-weight communication library for high-performance cloud networks. LibRIPC assumes heterogeneous cloud workloads running on closely-interconnected networks, with features similar to those found in present-day's HPC platforms. A key feature of libRIPC is that it offers separate transmit functions for the signaling- and data path. Early experiences with a libRIPC-powered Java web server indicate that libRIPC is easy to integrate and brings substantial speedup to cloud workloads.

Future work foremost includes evaluating libRIPC on other network architectures and workloads: a version for Ethernet/iWARP is underway, Blue Gene/P and PCIe versions are planned. Regarding workloads, we are currently exploring the benefits of libRIPC to Hadoop and to MongoDB, a NoSQL database written in C++. Conceptually, future work includes exploring how libRIPC could leverage group communication primitives that are often available on high-performance networks. Also, we plan to further explore and evaluate the reliability and reconfigurability of libRIPC in realistic cloud scenarios.

LibRIPC is open source and can be downloaded from http://github.com/jkehne/libripc.

References

[1] L. A. Barroso and U. Hölzle, *The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines*, ser. Synthesis Lectures on Computer Architecture. Morgan & Claypool, 2009.

[2] A. Zeichick, "How Facebook works," *Technology Review*, Jul. 2008.

[3] Y. Wang, X. Que, W. Yu, D. Goldenberg, and D. Sehgal, "Hadoop acceleration through network levitated merge," in *Proceedings of the 2010 International Conference on Supercomputing*, Seattle, WA, USA, Nov. 2011, p. 57.

[4] J. Jose, H. Subramoni, M. Luo, M. Zhang, J. Huang, M. W. ur Rahman, N. S. Islam, X. Ouyang, H. Wang, S. Sur, and D. K. Panda, "Memcached design on high performance RDMA capable interconnects," in *Proceedings of the 2011 International Conference on Parallel Processing*, Taipei, Taiwan, Sep. 2011, pp. 743–752.

[5] J. Appavoo, V. Uhlig, J. Stoess, A. Waterland, B. Rosenburg, R. Wisniewski, D. D. Silva, E. van Hensbergen, and U. Steinberg, "Providing a cloud network infrastructure on a supercomputer," in *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, Chicago, IL, USA, Jun. 2010, pp. 385–394.

[6] J. Stoess, J. Appavoo, U. Steinberg, A. Waterland, V. Uhlig, and J. Kehne, "A light-weight virtual machine monitor for Blue Gene/P," in *Proceedings of the 1st International Workshop on Runtime and Operating Systems for Supercomputers*, Tucson, AZ, Jul. 2011, pp. 3–10.

[7] D. Cohen, T. Talpey, A. Kanevsky, U. Cummings, M. Krause, R. Recio, D. Crupnicoff, L. Dickman, and P. Grun, "Remote direct memory access over the converged enhanced ethernet fabric: Evaluating the options," in *Proceedings of the 17th IEEE Symposium on High Performance Interconnects*, New York, NY, Aug. 2009, pp. 123–130.

[8] J. Byrne, J. Chang, K. T. Lim, L. Ramirez, and P. Ranganathan, "Power-efficient networking for balanced system designs: early experiences with PCIe," in *Proceedings of the 4th Workshop on Power-Aware Computing and Systems*, Cascais, Portugal, Oct. 2011, pp. 3:1–3:5.

[9] J. Liedtke, "Improving IPC by kernel design," in *Proceedings of the 14th Symposium on Operating System Principles*, Asheville, NC, USA, Dec. 1993, pp. 175–188.

[10] J. Pinkerton and E. Deleganes, "Direct Data Placement Protocol (DDP) / Remote Direct Memory Access Protocol (RDMAP) Security," RFC 5042 (Proposed Standard), Internet Engineering Task Force, Oct. 2007. [Online]. Available: www.ietf.org/rfc/rfc5042.txt

[11] R. Noronha, L. Chai, T. Talpey, and D. K. Panda, "Designing NFS with RDMA for security, performance and scalability," in *Proceedings of the 2007 International Conference on Parallel Processing*, Washington, DC, USA, Sep. 2007, pp. 49–56.

[12] Codehaus Foundation, "Jetty WebServer," jetty.codehaus.org/jetty/.

[13] Apache Software Foundation, "Hadoop," hadoop.apache.org/.

[14] JNA Team, "Java Native Access," github.com/twall/jna.

[15] iMatix Corp., "ZeroMQ," zeromq.com/.

[16] D. Goldenberg, M. Kagan, R. Ravid, and M. Tsirkin, "Zero copy sockets direct protocol over InfiniBand-preliminary implementation and performance analysis," in *Proceedings of the 13th IEEE Symposium on High Performance Interconnects*, Palo Alto, CA, Aug. 2005, pp. 128–137.

[17] S. Han, S. Marshall, B. Chun, and S. Ratnasamy, "MegaPipe: A new programming interface for scalable network I/O," in *Proceedings of the 10th Symposium on Operating Systems Design and Implementation*, Hollywood, CA, USA, Oct. 2012, pp. 135–148.

[18] F. MPI, "MPI: A message-passing interface," Oregon Graduate Institute School of Science & Engineering, Tech. Rep., 1994.

[19] S. Atchley, D. Dillow, G. Shipman, P. Geoffray, J. M. Squyres, G. Bosilca, and R. Minnich, "The common communication interface (CCI)," in *Proceedings of the 19th IEEE Symposium on High Performance Interconnects*, ser. HOTI '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 51–60.

[20] M. F. Kaashoek, R. van Renesse, H. van Staveren, and A. S. Tanenbaum, "FLIP: An internetwork protocol for supporting distributed systems," *ACM Transactions on Computer Systems*, vol. 11, no. 1, pp. 73–106, Apr. 1993.

[21] A. S. Tanenbaum, R. van Renesse, H. van Staveren, G. J. Sharp, and S. J. Mullender, "Experiences with the Amoeba distributed operating system," *Communications of the ACM*, vol. 33, no. 12, pp. 46–63, 1990.