RAMP: A Lightweight RDMA Abstraction for Loosely Coupled Applications

Babar Naveed Memon, Xiayue Charles Lin, Arshia Mufti, Arthur Scott Wesley
Tim Brecht, Kenneth Salem, Bernard Wong, Benjamin Cassell
University of Waterloo

Abstract

RDMA can be used to implement a shared storage abstraction for distributed applications. We argue that for loosely coupled applications, such an approach is overkill. For such applications, we propose RAMP, a much lighter weight alternative. RAMP uses RDMA only to support occasional coordination operations. We use a load balancing example to show that RAMP can effectively support such applications.

1 Introduction

Remote Direct Memory Access (RDMA) technology allows servers to directly access the memory of other servers in a cluster. This can significantly reduce network latency and eliminate network-related load on remote servers. The key question is how to take advantage of RDMA to build distributed applications. A common answer is that RDMA should be used to implement a cluster-wide shared-storage abstraction, such as a shared virtual address space or a database. Developers can then build applications that can access shared state from any server on the cluster.

The shared state approach is flexible, and RDMA-based shared state systems can be carefully engineered to achieve impressive performance [20, 8, 29]. However, although RDMA provides low-latency access to remote memory, access to local memory is still orders of magnitude faster. Furthermore, building a cluster-wide shared storage abstraction, even taking advantage of RDMA, necessarily introduces additional overheads. A shared-storage system must have some mechanism for locating data, and this imposes a level of indirection between the application and shared storage. In addition, since storage is shared, applications must have some means of synchronizing access. Most shared storage systems also only support storing strings or simple primitive types. Storing structured data requires serialization, which can introduce additional delays.

As a concrete example, access to a distributed hash table implemented on FaRM requires a few tens of microseconds, depending on the number of servers involved [8]. On one hand, this is impressive, especially considering that the hash table provides fault tolerance through replication and can be scaled out across servers. On the other hand, a simple single-server hash map, stored in local memory, has sub-microsecond access latency - orders of magnitude faster.

Many applications do not need the flexibility of fully shared data. Their data and workload are easily partitionable, and normally each server can handle its part of the workload using only local data. For example, `memcached` [11], which we discuss in more detail in Section 5, partitions keys across servers, each of which operates largely independently. However, servers in such applications may occasionally need to perform *coordination* operations, which require access to remote data. For example, servers may need to rebalance data and load because of time-varying hotspots, or the system may need to scale in or scale out to handle load fluctuations, or reconfigure itself in response to a server failure.

In this paper, we argue that shared state is overkill for such loosely coupled applications. Instead, we propose a lightweight model, called RAMP, that is well suited to support them. RAMP is lightweight in the sense that it does much *less* than shared state alternatives. Unless the application is coordinating (e.g., load balancing), RAMP stays out of the way, allowing the application to run at local memory speeds. The primary service provided by RAMP is low-impact, application-controlled state migration using RDMA. Loosely coupled applications use this when they coordinate. The low impact aspect of our approach is especially important, as the migration source may be overloaded due to a load imbalance.

RAMP enables a design point in between the common shared-memory and shared-nothing models for distributed applications. In Sections 4 and 5, we focus on

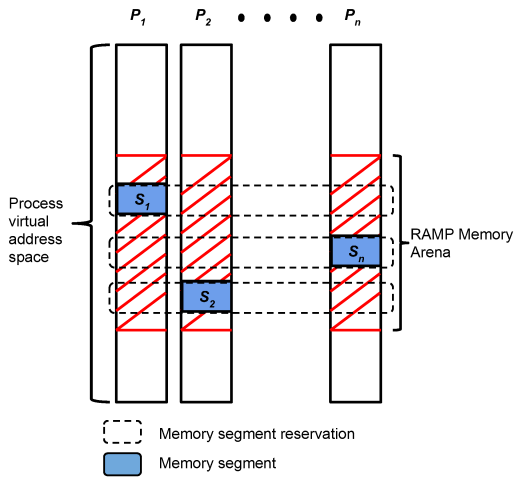

Figure 1: RAMP's Memory Model

how distributed applications can use RAMP. We consider a scenario in which the application migrates data containers between servers to shift load, and show that RAMP can perform such migrations with limited impact on application performance.

2 RAMP Overview

In this section we provide a high-level overview of RAMP's memory model and application interface. We had two goals in developing RAMP. First, each process in a loosely coupled distributed application should be able to directly access state through its own address space, without going through a platform API, and should be able to do so at local memory speeds. This is to ensure that processes perform well in the common case, when they are operating independently. Second, RAMP should provide a means of transferring data between processes to support coordination operations. RAMP should attempt to minimize the impact of such operations on the application's performance.

To provide these properties, RAMP implements the memory model illustrated in Figure 1. The figure illustrates the private virtual address spaces of n processes, each on a different server. RAMP reserves a memory arena covering a common range of virtual addresses in every process. Outside of the RAMP arena, each process is free to allocate and use memory independently of other processes in the system. Allocation and use of memory within the RAMP arena is controlled by RAMP.

Applications use the RAMP API, shown in Figure 2, to allocate *coordinated memory segments (S)* within the RAMP arena. When allocating a segment, the application specifies a unique segment identifier, which it can

```
//**** allocation ****
vaddr Allocate(size, segment_id);
Deallocate(segment_id)
//**** migration ****
Connect(process_id, segment_id);
Transfer(segment_id);
vaddr Receive(&segment_id,auto_pull);
//**** data transfer ****
Pull(vaddr, size);
//**** migration termination ****
Close(segment_id)
```

Figure 2: RAMP API

then use to migrate or deallocate the segment. Each segment occupies a fixed-length, contiguous range of virtual addresses within the RAMP arena. RAMP coordinates allocation across the processes so that a memory segment allocated by one process does not overlap with other segments allocated in any RAMP process.

Once it has allocated a segment, process P_i is free to read and write to virtual addresses within that segment, and we say that P_i *owns* the segment. RAMP does not interpose in any way on P_i's reads and writes to coordinated segments that it owns.

3 Migration

The key functionality provided by RAMP is the ability to *migrate* memory segments from one process (the *source*) to another (the *target*). Migrating a segment makes it available at the target and unavailable at the source. Since a migrated segment will map into the target's address space in the same place it was mapped into the source, intra-segment pointers remain valid across a migration. This allows applications to migrate segment-contained data structures *without serialization and deserialization*. We discuss this further in Section 4.

Ideally, migration would *atomically* and *instantaneously* transfer a segment from source to target. That is, the source would be able to read and write to the segment at local memory speeds up until a migration point, at which point the source would lose the ability to access the segment, and the target would gain it. The target would then be able to access the segment at local memory speed, and its reads would see the effects of all pre-migration writes.

In practice, of course, this is not possible. Segment migration in RAMP diverges from the ideal in two ways. First, there is a brief period during which neither the source nor the target can access the segment. RAMP's migration procedure is intended to keep this as short as possible (a few tens of microseconds), regardless of the

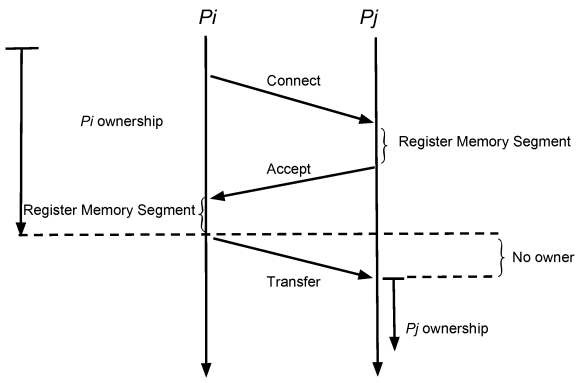

Figure 3: Transfer of Ownership in RAMP

size of the region being transferred. Second, there is some temporary performance impact as a result of migration. RAMP's design is intended to ensure negligible performance impact at the source, since a common application scenario is migration away from an already overloaded source. At the target, RAMP tries to minimize the impact of migration on memory access latencies.

To achieve this, RAMP separates transfer of ownership from transfer of data during segment migration. Ownership of the segment is transferred eagerly. The segment data are later pulled from the source by the target, either on-demand or asynchronously, in small chunks using one-sided RDMA reads. Separating data transfer from ownership transfer keeps the latter short. Migrating data gradually helps keep the access penalty low, and the use of one-sided reads allows RAMP to migrate the data without involving the CPU at the source. In Sections 3.1 and 3.2, we describe ownership transfer and data transfer in RAMP in more detail.

3.1 Ownership Transfer

Figure 3 illustrates RAMP's ownership transfer protocol. Ownership transfer requires that the source call `Connect` and `Transfer`, and that the target call `Receive` to accept ownership. When the source calls `Connect`, RAMP establishes a reliable RDMA connection from the source to the target, registers S for RDMA at the source (pinning S in physical memory), and sends (via the RDMA connection) a *Connect* message specifying the starting address and size of S to the target server. At the target, RAMP maps S into the virtual address space and registers S for RDMA at the target. Finally, the target responds to the source with an (Accept) message, indicating that it is prepared to receive ownership of S.

This connection process is relatively expensive, as it involves RDMA connection setup, a message round trip, and RDMA memory registration on both sides. However, the source retains ownership of S during this entire connection process, and can continue to read and write to it. It introduces a lag at the source between deciding to migrate and migrating, but it does not impact memory access times during that lag.

Actual transfer of ownership occurs when the source calls `Transfer`, at which point it loses the right to read from or write to S. (RAMP enforces this by `mprotecting` S at the source.) The source sends a `Transfer` message via the RDMA connection to notify the target that it now owns S. On receipt of this message at the target, `Receive` returns with the address and ID of the incoming segment. The target now has the ability to read and write to addresses in S, although S's contents have not yet been transferred.

3.2 Pulling Data

In RAMP, data are pulled under the direction of the application at the target site. RAMP supports both *implicit pulls* and *explicit pulls*. implicit pulls by setting the `auto_pull` flag when When implicit pulling is enabled, RAMP automatically pulls pages of S from the source on demand, when they are first accessed by the application at the target. RAMP implements this by `mprotecting` S when ownership is transferred, and pulling data on protection faults. In addition, the application can explicitly prefetch any part of memory segment using RAMP's `Pull` interface. All pulls are implemented as one-sided RDMA reads from the source. While the target is pulling the contents of S, S is registered for RDMA and mapped in to the address at both the target and the source, although only the target owns S and is able to access it. When the application is finished pulling S to the target site, it uses `Close` to end the migration process.

4 Migratable Containers

One way for applications to use RAMP is to place *self-contained* data structures within a migratable memory segment. These structures can then be migrated between servers by migrating the underlying segment. There is *no need to serialize or deserialize the structure* when it migrates, since a segment's position in the virtual address space of the target is the same as its position at the source. By doing this, applications can lift the level of abstraction for migration, from migrating memory segments to migrating data structures.

To illustrate this usage pattern, we implemented *migratable containers* for C++ applications. A migratable container is a standard C++ standard template library (STL) container (e.g, a hash map) with a few modifications. First, a migratable container uses a custom C++ scoped memory allocator to ensure that all

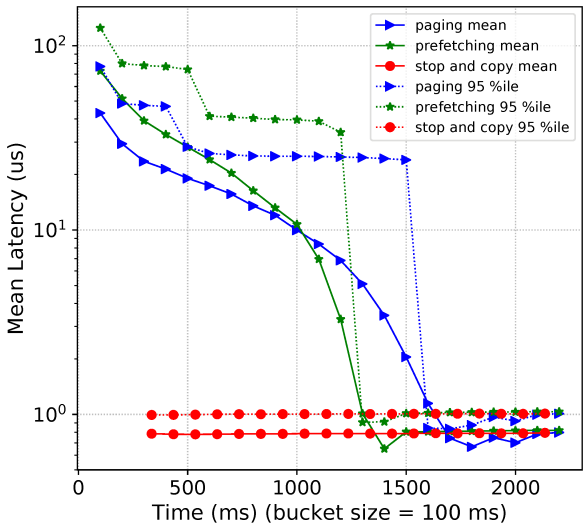

Figure 4: Post-Migration Container Access Latency

memory allocated for the container and its contents lies within a segment. Second, the container is equipped with an additional constructor, which is used to initialize the container at the target after migration. Finally, the container's normal interface is supplemented with additional methods analogous to those in Figure 2, to provide container migration capability. Except during migrations, migratable containers are identical to their non-migratable cousins, and run at their original local memory speeds.

To migrate a container, the application uses the container's `Connect` and `Transfer` methods, which work by transferring the underlying segment. At the target, the application immediately begins using the container as soon as it receives ownership of the segment. Our implementation uses RAMP's implicit paging to demand pull chunks of segment data from the source as the container is used at the target. Our implementation also uses RAMP's prefetching mechanism.

To illustrate the behavior of migratable containers, we created a migratable C++ unordered map in a 256 MB RAMP memory segment, populated it with about one million entries (8 byte keys, 128 byte values), and then migrated it to a second server across a 10 Gb/s network. At the target, a single application thread begins performing "get" operations in a tight loop immediately after receiving the incoming container. We performed three versions of this experiment. In the first, implicit pulling is used to pull the container data on demand. In the second, implicit pulling is supplemented by sequential prefetching via `Pull`. In the third, which we refer to as stop-and-copy, the container uses a single large `Pull` request to pull all of its data before allowing any container operations. Figure 4 illustrates the latency of post-migration container operations as a function of time, with 0 representing the point at which the target receives ownership of the container. We show the mean and 95th percentile latency, measured over 100 ms windows.

Using the stop-and-copy approach, all container accesses are performed at local memory speeds, with tail latency at about 1 μs, and sub-microsecond mean access times. However, the container is effectively unavailable for several hundred milliseconds after ownership is transferred, while the data are pulled, and this unavailability window would grow in proportion to the container size. With implicit demand paging, the container is available immediately, but there is a period of about 1.5 seconds during which container access latencies are elevated. Tail latencies remain below 50 μs during most of this period. Adding prefetching pulls the container data more quickly, but results in higher latencies immediately after migration. This is because our current implementation uses a single RDMA connection between target and source, causing prefetches to delay demand pulls. Once the underlying segment has been pulled, container memory accesses are completely local, and the container again provides local memory speeds, with sub-microsecond access latencies.

For larger containers, we expect a longer window of elevated access latencies, but tail latencies should not be higher than those shown in Figure 4. Our current prefetching implementation is not optimized, but by issuing asynchronous prefetch requests on a separate connection, we expect to be able to prefetch data at close to the full network bandwidth.

5 Using RAMP

We built a loosely coupled application called *rcached* as a vehicle for evaluating RAMP. `rcached` is a simple drop-in replacement for `memcached`, a widely used distributed in-memory key-value storage service. `rcached` hash partitions the key space, and stores each partition in a migratable C++ STL hash map. Each `rcached` server is responsible for a subset of the partitions. Unlike `memcached`, `rcached` servers can migrate partitions amongst themselves to support reconfigurations (e.g., scale-in, scale-out) or load balancing. `rcached` is not as heavily engineered as memcached, but under light loads its performance is similar. On our servers, a lightly loaded `memcached` server with four request handling threads has mean request latency of about 35 μs, while a similarly configured `rcached` server has a mean request latency of about 50 μs.

Our load-balancing experiment used a cluster of 4 `rcached` servers, on which we stored 40 million keys, each associated with a 128 byte value, hash partitioned

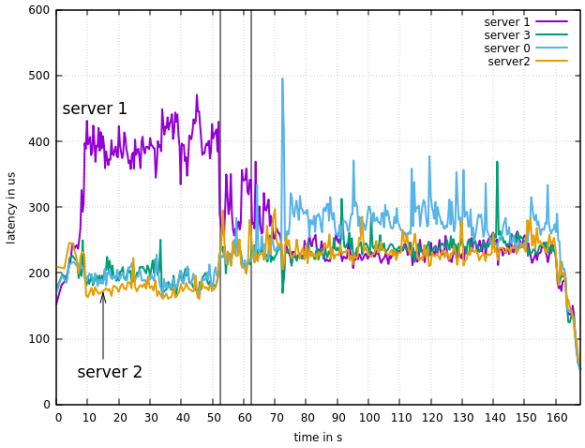

Figure 5: `rcached` Request Latencies, With Two Migrations

into 128 partitions. 100 closed-loop clients issued "get" requests, using a Zipf distribution (parameter 0.99) over the key space and no think time. Total request throughput averaged over 380000 requests per second over the experiment. Because of request skew and the imperfect distribution of partitions to servers resulting from consistent hashing, the request load on the servers is skewed, with server 1 receiving the most requests in our configuration. After allowing the system to warm up, we used RAMP to migrate two heavily loaded partitions from server 1 to server 2, to better balance the load.

Figure 5 shows the mean client-side request latency in this system, averaged over 40000-request windows, and broken down by the server that handled the request. The two vertical lines indicate the approximate times at which partition migrations occurred. Because of our closed-loop setting, all other servers see additional load when server 1's load drops after the migration. Server 2 shows short spikes in 95th percentile latencies (not shown) at the time of the migrations, as anticipated, although they are no worse than the normal latency variations that occur due to random short term load fluctuations. In short, `rcached` is able to load balance effectively, with minimal interruption, with a few RAMP container migrations.

6 Related Work

As we noted in Section 1, RDMA can be used to implement a cluster-wide shared-storage abstraction that allows fine-grained "anything from anywhere" access. FaRM [8] provides a cluster-wide fault-tolerant global shared address space, which applications access using FaRM-serialized transactions. Similarly, Nam-DB [4, 29] and Tell [20] use RDMA to provide transactional access to a cluster-wide shared database or key-value store [5, 14, 23, 28]. RAMP operates a different point in the design space, offering much *less* functionality than shared-state alternatives. RDMA has also been used to implement high-performance remote procedure call mechanisms for shared-nothing applications [15, 27], and to build efficient mechansisms for data exchange to support distributed joins and similar operations [1, 3, 12, 19]. Unlike that work, RAMP focuses on occasional exchanges of large amounts of information, while minimizing application impact. Rocksteady [17] uses RDMA to implement application-initiated data migration in the context of RamCloud [24]. Like RAMP, Rocksteady transfers ownership eagerly and data lazily. Rocksteady does not make use of one-sided RDMA reads for data migration, since a RamCloud tablet (the unit of migration) may be scattered in memory.

Live migration of state has also explored in a variety of contexts, including distributed database management systems [2, 7, 9, 10, 21, 26], and virtual machine cloning [18, 22] and migration [6, 13]. All of these systems use some combination of pulling data from the target (like RAMP) and pushing it from the source to accomplish migration. However, none are designed to take advantage of RDMA. RAMP is also related to distributed shared memory (DSM) systems [25], although RAMP is simpler in that it allows only one process to access a memory segment at any time.

Khrabrov and de Lara [16] describe a system that allows applications to create structured objects in a shared global address space and then transfer them between servers in a cluster. As is the case with RAMP containers, transfer can be accomplished without serializing and deserializing the objects. However, this approach is Java-specific, and objects are immutable once transferred. It does not make use of RDMA, although it could.

7 Conclusion

RAMP provides a lightweight RDMA abstraction that is intended for loosely coupled distributed applications. Instead of shared storage, RAMP provides fast transfer of memory segments between servers. We showed that RAMP can be used to build migratable data structures than can be transferred between servers with little performance impact. We demonstrated the use of RAMP for load balancing in rcached, a memcached-compatible distributed in-memory key-value cache.

8 Acknowledgments

This work was supported by Huawei Technologies, under Research Agreement YB2015120103.

References

[1] ALONSO, G., AND FREY, P. W. Minimizing the hidden cost of RDMA. In *Proc. Int'l Conf. on Distributed Computing Systems* (2009), pp. 553–560.

[2] BARKER, S., CHI, Y., MOON, H. J., HACIGÜMÜŞ, H., AND SHENOY, P. "Cut Me Some Slack": Latency-aware live migration for databases. In *Proc. EDBT* (2012), pp. 432–443.

[3] BARTHELS, C., LOESING, S., ALONSO, G., AND KOSSMANN, D. Rack-scale in-memory join processing using RDMA. In *Proc. SIGMOD* (2015), pp. 1463–1475.

[4] BINNIG, C., CROTTY, A., GALAKATOS, A., KRASKA, T., AND ZAMANIAN, E. The end of slow networks: It's time for a redesign. *Proc. VLDB Endow. 9*, 7 (Mar. 2016), 528–539.

[5] CASSELL, B., SZEPESI, T., WONG, B., BRECHT, T., MA, J., AND LIU, X. Nessie: A decoupled, client-driven key-value store using RDMA. *IEEE Transactions on Parallel and Distributed Systems 28*, 12 (Dec 2017), 3537–3552.

[6] CLARK, C., FRASER, K., HAND, S., HANSEN, J. G., JUL, E., LIMPACH, C., PRATT, I., AND WARFIELD, A. Live migration of virtual machines. In *Proc. NSDI* (2005), pp. 273–286.

[7] DAS, S., NISHIMURA, S., AGRAWAL, D., AND EL ABBADI, A. Albatross: Lightweight elasticity in shared storage databases for the cloud using live data migration. *Proc. VLDB Endow. 4*, 8 (May 2011), 494–505.

[8] DRAGOJEVIĆ, A., NARAYANAN, D., HODSON, O., AND CASTRO, M. Farm: Fast remote memory. In *Proc. NSDI* (2014), pp. 401–414.

[9] ELMORE, A. J., ARORA, V., TAFT, R., PAVLO, A., AGRAWAL, D., AND EL ABBADI, A. Squall: Fine-grained live reconfiguration for partitioned main memory databases. In *Proc. SIGMOD* (2015), pp. 299–313.

[10] ELMORE, A. J., DAS, S., AGRAWAL, D., AND EL ABBADI, A. Zephyr: Live migration in shared nothing databases for elastic cloud platforms. In *Proc. SIGMOD* (2011), pp. 301–312.

[11] FITZPATRICK, B. Distributed caching with memcached. *Linux journal 2004*, 124 (2004).

[12] FREY, P. W., GONCALVES, R., KERSTEN, M., AND TEUBNER, J. A spinning join that does not get dizzy. In *Proc. ICDCS* (2010), pp. 283–292.

[13] HUANG, W., GAO, Q., LIU, J., AND PANDA, D. K. High performance virtual machine migration with RDMA over modern interconnects. In *Proc. IEEE Int'l Conf. on Cluster Computing* (2007), pp. 11–20.

[14] KALIA, A., KAMINSKY, M., AND ANDERSEN, D. G. Using RDMA efficiently for key-value services. *SIGCOMM Comput. Commun. Rev. 44*, 4 (Aug. 2014), 295–306.

[15] KALIA, A., KAMINSKY, M., AND ANDERSEN, D. G. FaSST: Fast, scalable and simple distributed transactions with two-sided (RDMA) datagram RPCs. In *Proc. OSDI* (2016), pp. 185–201.

[16] KHRABROV, A., AND DE LARA, E. Accelerating complex data transfer for cluster computing. In *Proc. HotCloud* (2016), pp. 40–45.

[17] KULKARNI, C., KESAVAN, A., ZHANG, T., RICCI, R., AND STUTSMAN, R. Rocksteady: Fast migration for low-latency in-memory storage. In *Proc. SOSP*, pp. 390–405.

[18] LAGAR-CAVILLA, H. A., WHITNEY, J. A., BRYANT, R., PATCHIN, P., BRUDNO, M., DE LARA, E., RUMBLE, S. M., SATYANARAYANAN, M., AND SCANNELL, A. Snowflock: Virtual machine cloning as a first-class cloud primitive. *ACM Trans. Comput. Syst. 29*, 1 (2011), 2:1–2:45.

[19] LIU, F., YIN, L., AND BLANAS, S. Design and evaluation of an RDMA-aware data shuffling operator for parallel database systems. In *Proc. EuroSys* (2017), pp. 48–63.

[20] LOESING, S., PILMAN, M., ETTER, T., AND KOSSMANN, D. On the design and scalability of distributed shared-data databases. In *Proc. SIGMOD* (2015), pp. 663–676.

[21] MINHAS, U. F., LIU, R., ABOULNAGA, A., SALEM, K., NG, J., AND ROBERTSON, S. Elastic scale-out for partition-based database systems. In *Proc. ICDE Workshops, Workshop on Self-Managing Database Systems* (2012), pp. 281–288.

[22] MIOR, M. J., AND DE LARA, E. Flurrydb: a dynamically scalable relational database with virtual machine cloning. In *Proc. SYSTOR* (2011).

[23] MITCHELL, C., GENG, Y., AND LI, J. Using one-sided RDMA reads to build a fast, CPU-efficient key-value store. In *Proc. USENIX ATC* (2013), pp. 103–114.

[24] OUSTERHOUT, J., GOPALAN, A., GUPTA, A., KEJRIWAL, A., LEE, C., MONTAZERI, B., ONGARO, D., PARK, S. J., QIN, H., ROSENBLUM, M., RUMBLE, S., STUTSMAN, R., AND YANG, S. The RAMCloud storage system. *ACM Trans. Comput. Syst. 33*, 3 (Aug. 2015), 7:1–7:55.

[25] PROTIC, J., TOMASEVIC, M., AND MILUTINOVIC, V. Distributed shared memory: Concepts and systems. *IEEE Parallel Distrib. Technol. 4*, 2 (June 1996), 63–79.

[26] SCHILLER, O., CIPRIANI, N., AND MITSCHANG, B. Prorea: Live database migration for multi-tenant RDBMS with snapshot isolation. In *Proc. EDBT* (2013), pp. 53–64.

[27] SU, M., ZHANG, M., CHEN, K., GUO, Z., AND WU, Y. RFP: When RPC is faster than server-bypass with RDMA. In *Proc. EuroSys* (2017), pp. 1–15.

[28] WEI, X., SHI, J., CHEN, Y., CHEN, R., AND CHEN, H. Fast in-memory transaction processing using RDMA and HTM. In *Proceedings of the 25th Symposium on Operating Systems Principles* (New York, NY, USA, 2015), SOSP '15, ACM, pp. 87–104.

[29] ZAMANIAN, E., BINNIG, C., HARRIS, T., AND KRASKA, T. The end of a myth: Distributed transactions can scale. *Proc. VLDB Endow. 10*, 6 (Feb. 2017), 685–696.