RAMP: RDMA Migration Platform

by

Babar Naveed Memon

A thesis
presented to the University of Waterloo
in fulfilment of the
thesis requirement for the degree of
Masters of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2018

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

Remote Direct Memory Access (RDMA) can be used to implement a shared storage abstraction or a shared-nothing abstraction for distributed applications. We argue that the shared storage abstraction is overkill for loosely coupled applications and that the shared-nothing abstraction does not leverage all the benefits of RDMA. In this thesis, we propose an alternative abstraction for such applications using a shared-on-demand architecture, and present the RDMA Migration Platform (RAMP). RAMP is a lightweight coordination service for building loosely coupled distributed applications. This thesis describes the RAMP system, its programming model and operations, and evaluates the performance of RAMP using microbenchmarks. Furthermore, we illustrate RAMPs load balancing capabilities with a case study of a loosely coupled application that uses RAMP to balance a partition skew under load.

Acknowledgements

I want to begin by thanking my supervisor, Prof. Ken Salem, for his continuous support, guidance, and advice throughout my graduate studies and research. Ken has helped me understand the database and systems research landscape and his research style and methodology has taught me how to approach research problems. I am grateful for everything I have learned under his mentorship.

I express my sincerest gratitude to Prof. Bernard Wong and Prof. Tim Brecht for their continuous guidance and feedback which has been instrumental in building this work. They also graciously served on my thesis committee. I also want to extend my gratitude to Xiayue Charles Lin, Arshia Mufti, Scott Wesley, and Benjamin Cassel. Their collaboration and help have been indispensable in completing my thesis work.

A special shout-out to my colleague Siddhartha Sahu for helping me formulate my ideas and taking the time to proofread my thesis . I also thank my lab mates in the data systems group, Chathura Kankanamge and Amine Mhedhbi for patiently listening to the various problems that came up throughout my graduate studies and providing helpful suggestions. I extend my gratitude to my friends in Waterloo, Akanksha Mahajan and Tarun Patel, and to my friends spread across the world, Farah Zafar, Danial Azam, and Nina Anklesaria, for supporting me in all my endeavours. You hold a special place in my heart.

Finally, I am deeply grateful to my family for their unconditional love and support throughout my studies. This thesis would not be possible without the constant encouragement from my mother Mussarat Naveed, my sister Iqra Naveed and brother-in-law Mursal Shamsi who kept me energized with their constant supply of love, food and care, and last but definitely not the least, my siblings Zohaib Naveed, Danish Naveed, and Ahmed Shoaib whose entertaining conversations from the other side of the globe made me feel like I was home with them. Thank you and I love you all.

Dedication

This thesis is dedicated to my mother, Mussarat Naveed, for her love and countless sacrifices for my education and to my late father, Naveed Ghulam Qadir, who taught me to be passionate, kind, and encouraged me to pursue my dreams.

Table of Contents

List of Figures

Chapter 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 network-related bottlenecks. However, exploiting RDMA requires a significant redesign of cluster applications. Therefore, the key question is how we should leverage RDMA to build distributed applications. One common strategy is to use RDMA to implement a cluster-wide shared memory abstraction, such as a shared virtual address space or a database. Using this strategy, developers build applications that can access shared state from any server in the cluster. Another common strategy is to use RDMA to provide high performance remote procedure calls (RPC) or message passing. This strategy allows developers to build shared-nothing applications with high-performance inter-process communication.

The shared state approach is very flexible, and RDMA-based shared state systems can be carefully engineered to achieve impressive performance [34, 14, 59]. 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 memory abstraction to take advantage of RDMA introduces additional overheads. A shared memory system must have some mechanism for locating data on every application access and this imposes a level of indirection between the application and shared memory. In addition, since memory is shared, applications must have some means of synchronizing access. Most shared memory 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 [14]. 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 hashmap, stored in local memory, has sub-microsecond access latency, which is orders of magnitude faster. Therefore, even with RDMA, maximizing local access is preferred.

RDMA can be used to build a fast message passing layer. Such systems can also be engineered to achieve great performance [27]. However such systems do not utilize remote server bypass, which is RDMA's most attractive property. They are also typically designed to support small message sizes. Movement of large amounts of structured data requires serialization and deserialization, which adds additional overhead.

Many applications do not need the flexibility of fully shared data but require a richer mechanism than RPCs for sharing data. The data and workload of such applications are easily partitionable, and normally each server can handle its part of the workload using only local data. However, servers 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 hot spots or the system may need to scale in or scale out dynamically to handle time-varying load intensity, or it may reconfigure itself in response to a server failure.

In this thesis, we argue that shared state is overkill for such loosely coupled applications and the RPC abstraction is insufficient to utilize the benefits of RDMA. Instead, we present a lightweight shared-on-demand model, called RAMP, that is well suited to support such applications. RAMP is lightweight in the sense that it does much *less* than shared state alternatives and yet provides an efficient mechanism to move large amounts of data. 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 a low-impact and application-controlled state migration using RDMA. Loosely coupled applications use this service when they require coordination between servers. The low impact aspect of our approach is especially important because the migration source may be overloaded due to a load imbalance. RAMP enables a design point in between the shared memory and the shared-nothing models which benefits from RDMA's remote server bypass and eliminates access synchronization and data serialization costs.

The main contributions of this thesis are:

1. The design of RAMP.
2. An implementation of the RAMP model.
3. A performance evaluation of RAMP's operations with microbenchmarks and an application based case study.

4. An analysis of fault tolerance in RAMP, and extensions to the RAMP design to address failures.

The rest of this thesis is organized as follows: Chapter 2 provides RDMA background and highlights why applications require redesign to utilize it. Chapter 3 presents RAMP's memory model, describes its API, and illustrates how the API can be utilized by applications. Chapter 4 presents the fault tolerance mechanisms in RAMP. Chapter 5 presents the RAMP performance evaluation as well as an example of the use of RAMP in an application. Chapter 6 presents related work and Chapter 7 concludes the thesis while exploring potential future work.

Chapter 2

RDMA Background

This chapter provides an overview of Remote Direct Memory Access (RDMA). It describes the motivation for RDMA, compares RDMA with traditional TCP sockets, and highlights the performance gains possible with RDMA. Furthermore, this chapter discusses the programming model for RDMA and explains why there is a need to redesign applications that want to take advantage of RDMA primitives.

2.1 The Problem with Traditional Sockets

The Transmission Control Protocol (TCP) and Internet Protocol (IP) (denoted as TCP/IP together) and the traditional socket network programming model have been studied thoroughly over the years. In this section, we discuss why the socket programming model over TCP/IP is unable to provide the best possible performance for network communications.

When transmitting data using traditional sockets over TCP/IP, the data has to pass through multiple layers. It goes first from user space into a temporary socket buffer in kernel space and then through the TCP/IP layer (requiring additional processing to add appropriate headers), and finally to the network interface controller (NIC) using a direct memory access (DMA) operation. When a packet is received, the process is reversed. The data is moved from the NIC to kernel space with a DMA copy followed by a copy to user space when the application performs a read operation. Frey et al. [22] show that the main reason for performance degradation in the socket abstraction comes from CPU overhead due to the indirect data placement for network transfers, as shown in Figure 2.1. In addition to CPU overhead, Frey et al. [22] also show that the intermediate copies cause

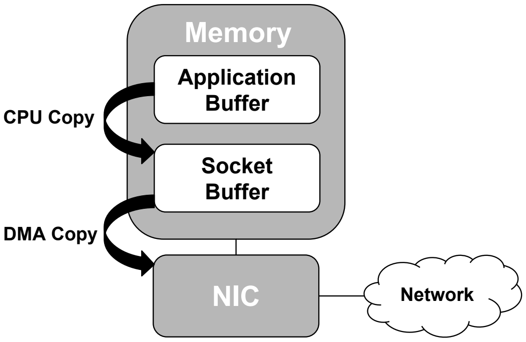


Figure 2.1: Indirect Data Placement in Traditional Sockets From [22]

performance degradation due to high memory bus traffic and numerous context switches occurring on kernel interrupts. The research community and industry have worked on optimizations, such as TCP offloading [11], in order to mitigate performance degradation in sockets. However, indirect data placement remains the primary bottleneck in the socket abstraction. RDMA circumvents this issue by using a zero-copy mechanism to transfer data.

2.2 Remote Direct Memory Access (RDMA)

RDMA is a network protocol that allows for a zero-copy transfer or direct data placement (DDP) when transferring data over a network. The transport protocol is implemented directly in the hardware drivers and exposed at the user-level. RDMA-enabled network controllers (RNICs) transfer memory directly from the user space of a process on one server to the user space of a process on the target server, avoiding intermediate copies in the kernel space. To enable DDP in RDMA, applications are required to explicitly manage memory buffers and register them with the network card. The registration process pins the virtual memory to physical memory, assigns it an identifier and maps it to the network card. Servers having the identifiers to this memory block may access it as long as it has not been de-registered. The latest RNICs are capable of completing requests in 1-3 microseconds at a bandwidth of 100 Gbps, thus providing more than an order of magnitude better performance than TCP/IP [13].

As shown in Figure 2.2, there are multiple variants of RDMA with varying underlying

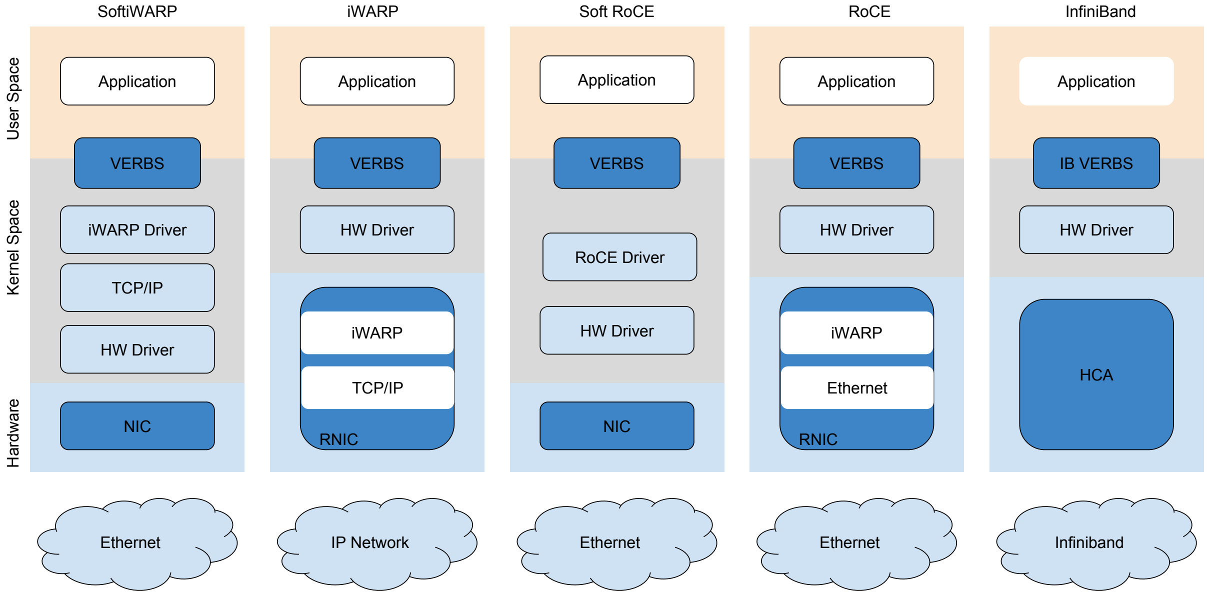
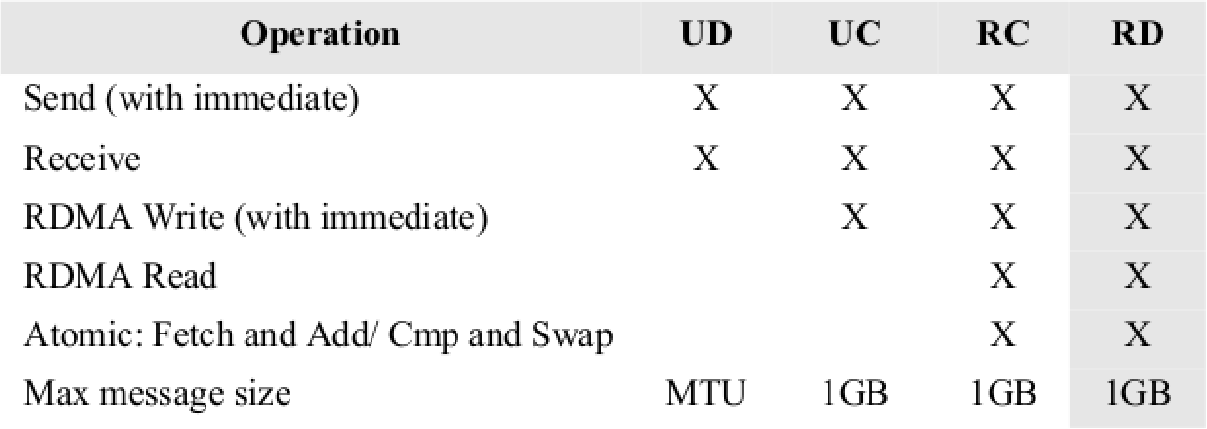


Figure 2.2: RDMA Providers

hardware. They all provide a verbs interface as defined in the RDMA specification. One such variant is InfiniBand (IB) [1], a network fabric used for high-performance computing for enterprise data centres. It is designed to provide high throughput, low latency, quality of service guarantees, and failover mechanisms. It provides an RDMA interface using Host Channel Adaptors (HCA) via a specialized NIC and an IB Verbs interface. Other variants include Internet Wide Area RDMA (iWARP) [9] and RDMA over Converged Ethernet (RoCE) [2] that provide RDMA interfaces over the IP and the Ethernet layer respectively using specialized NICs. SoftiWARP [3] and Soft RoCE [4], on the other hand, are software variants of iWARP and RoCE. They are kernel modules designed to inter-operate with remote hardware-based iWARP and RoCE. This enables a commodity NIC to interact with RDMA traffic at the software level. In essence, RDMA is an alternative network protocol with a verbs interface that provides the key feature of a zero copy transfer of data between remote servers over the network.

RDMA offers multiple types of communication mechanisms using Queue Pairs (QP). A QP is a socket equivalent in RDMA and is described in Section 2.3. The communication mechanisms offered are:

- Reliable Connected (RC)



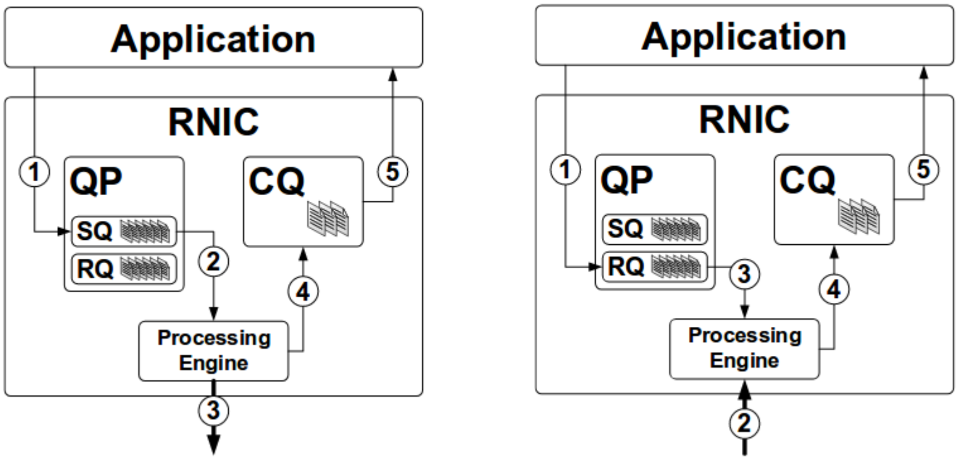
Operation	UD	UC	RC	RD
Send (with immediate)	X	X	X	X
Receive	X	X	X	X
RDMA Write (with immediate)		X	X	X
RDMA Read			X	X
Atomic: Fetch and Add/ Cmp and Swap			X	X
Max message size	MTU	1GB	1GB	1GB

Figure 2.3: RDMA Operations [35]

- Reliable Datagram (RD)
- Unreliable Connected (UC)
- Unreliable Datagram (UD)

The RC and UD protocols are similar to TCP and UDP in the socket programming model. The RD protocol is not supported by the verbs interface. Figure 2.3 shows a summary of different QPs offered in RDMA along with the operations they support.

The RC protocol supports reliable in order delivery of large data transfers. It supports both paired transfer (two-sided operations) for sending and receiving data and un-paired transfer (one-sided operations) involving an RDMA read or an RDMA write. Two-sided operations follow a message passing model and require the active involvement of CPUs on both the sending and the receiving servers. After establishing a connection, the sending process registers a memory region and sends data using the SEND verb to the target destination. The receiving process, however, needs to register a memory region and post it with the RNIC using the RECEIVE verb for storing the incoming data. The RNIC then places this data in the posted memory region and adds it to the receive queue. One-sided read and write operations bypass the remote CPU and operate directly on remote memory using a DMA. In order to enable one-sided read and write operations, the host process establishes connections with the remote processes and registers memory regions with the RNIC specifying permissions for one-sided operations. The host process then exchanges memory region and access key information with remote processes. Using these access keys and memory region information, together with the starting address and size, remote



(a) Post Send

(b) Post Receive

Figure 2.4: RDMA Programming Model [22]

processes may perform read and write operations directly without the involvement of the host CPU. The RC protocol also supports atomic *compare-and-swap* and *fetch-and-add* operations for 64-bit memory regions that work in the same fashion as RDMA read and write operations.

The UC and UD protocols are unreliable and unordered transport protocols. There is no guarantee that the receiving node will receive incoming messages. Corrupted and out of sequence packets are simply dropped. The UC protocol supports large message transfers using SEND and RECEIVE operations similar to RC. The UD protocol only supports datagrams up to 4 KB but it does not require a connection to be established between endpoints and it supports multicast.

2.3 RDMA Programming Model and the Need for Redesign

RDMA's queue-based asynchronous programming model differs significantly from traditional socket programming. Using the verbs interface, applications interact with the RNIC and request operations. The communication paradigm is based on a send queue (SQ) and receive queue (RQ) as shown in Figure 2.4. These queues are together known as a queue pair (QP). A completion queue (CQ) is also attached to every QP. The application creates work requests (WR) and posts them on the QP. Outgoing requests are placed on the SQ while incoming requests are posted on the RQ. As work requests are completed, a completion queue element (CQE) is posted on the CQ. Applications periodically poll the CQ for notifications. Alternatively, applications poll the local memory regions directly. Further details can be found in a study conducted by Frey et al. [22].

The stark difference between RDMA and socket programming means that a significant porting effort is required to move socket-based applications to RDMA. This issue is exacerbated by the fact that RDMA requires users to explicitly manage network buffers and poll for request completions. The difference in programming models and explicit buffer management introduces hidden costs to using RDMA. When handled without careful consideration, they may negate the performance benefits that RDMA brings.

One such hidden cost is *memory registration* latency. To illustrate this cost, we performed an experiment to measure the latency to register and de-register memory with a RNIC. We set up a server to allocate 10 memory regions of a particular size (configurations for the server and RNIC are described in Chapter 5). These memory regions are registered with the RNIC one by one using a single thread and then de-registered. The registration and de-registration periods of the 10 memory regions are measured separately. We calculate the mean and the standard deviation of our latency measurements. The results (shown in Figure 2.5) demonstrate that memory registration and de-registration with the RNIC have high latency and therefore should be handled outside the performance critical path of execution. Note that the standard deviations are often too small to be visible in the figure.

Frey et al. [20] presents several other hidden RDMA costs, such as increased connection setup time and complex RDMA object management. Therefore, it is important that special attention is given to software design when using RDMA.

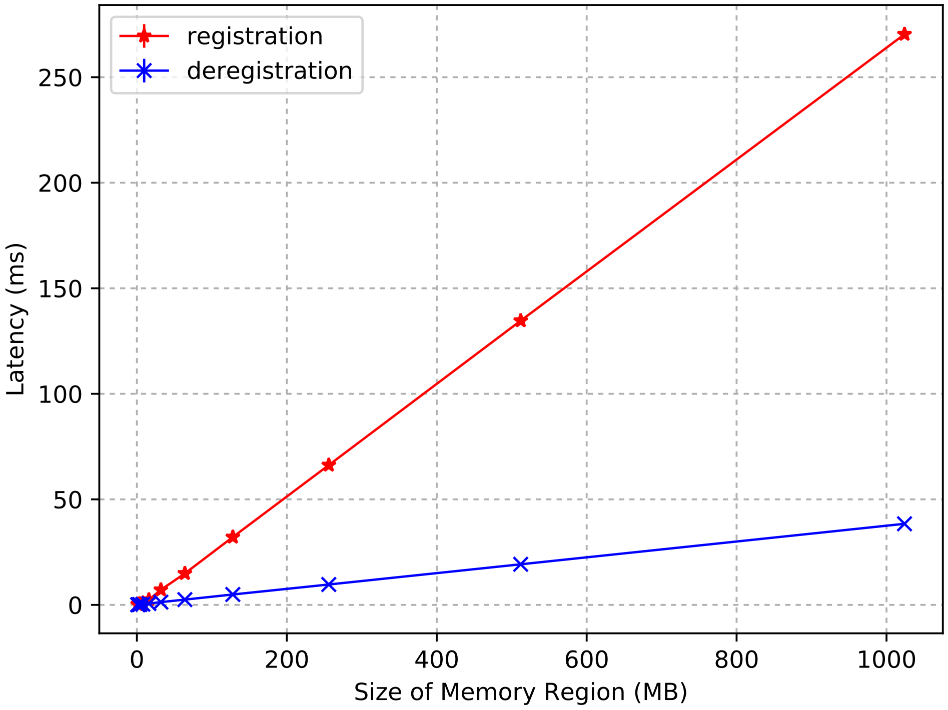


Figure 2.5: Memory Registration Latency

Chapter 3

RAMP: RDMA Migration Platform

This chapter provides an overview of RAMP. It describes the system architecture and the RAMP programming model. Furthermore, this chapter describes RAMP operations and their internal workings.

3.1 System Overview

RAMP is a migration platform for loosely coupled data-intensive applications. It provides a programming model that leverages RDMA to seamlessly migrate *memory segments* between nodes in a cluster. A memory segment is a block of contiguous virtual memory allocated through the RAMP interface. RAMP uses a coordinated allocator to ensure that virtual addresses of a memory segment are reserved across the cluster. This coordinated allocation permits data migration without serialization or deserialization. RAMP allows applications to create, migrate, and delete memory segments while ensuring that system properties are upheld. RAMP is designed to minimize memory access latencies during segment migrations. In order to achieve this, RAMP decouples the ownership of a memory segment from the movement of data during migration. Ownership is transferred using migration operations described in Section 3.4.1 and the data is moved across the cluster using one-sided RDMA read operations (Section 3.4.2).

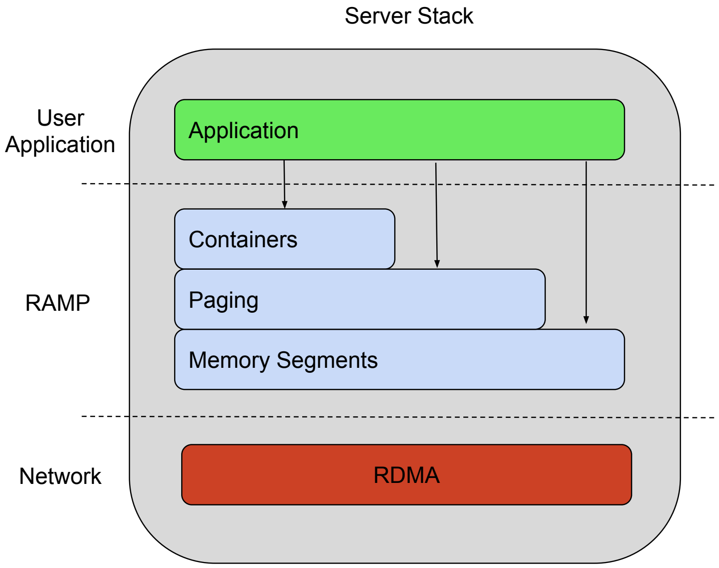


Figure 3.1: RAMP Architecture

3.2 Architecture

Figure 3.1 shows the RAMP software stack. At the base, RAMP uses Reliable Connected (RC) RDMA for its network operations. RDMA connections, buffer management for message exchanges, and memory registration are handled internally by RAMP. The RAMP system has three components: a *memory segment layer*, a *paging layer*, and a *container layer*. Each layer builds on the functionality and API of the underlying layer. Applications can use the system through any layer.

The memory segment layer provides the programming model and the base RAMP API. It allows applications to create, migrate, and delete memory segments. The paging layer automates and customizes data movements during migrations. After an ownership transfer, the paging layer pulls small chunks of the memory segment on demand, keeping data access latency low. The containers layer allows applications to create standard template library (STL) data structures stored within a memory segment. Applications leverage the containers layer to utilize RAMP migration functionality with their existing data model with minimal porting effort.

3.3 Memory Segment Layer

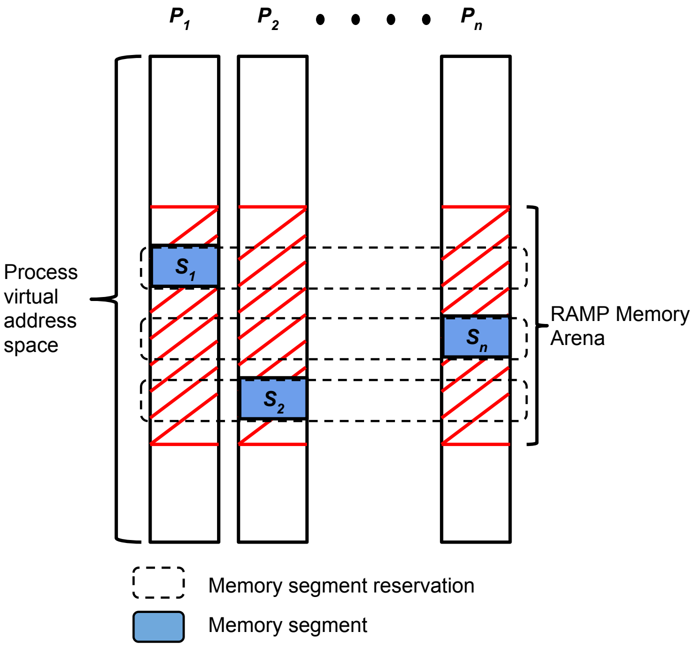


Figure 3.2: Coordinated Allocator Reservation

The RAMP programming model is based on *coordinated memory segments*. A memory segment is a contiguous range of virtual addresses and provides the basic unit of abstraction in RAMP. A coordinated memory segment is a memory segment that is reserved across all configured servers in RAMP. All operations are performed against a memory segment and RAMP ensures that only a single process owns a memory segment at any time. Furthermore, only the segment owner is allowed to read or write to the virtual addresses within a memory segment, i.e., single reader and single writer semantics are enforced.

RAMP's memory model is illustrated in Figure 3.2. The figure shows the virtual address spaces of n processes, where each process represents a different server. RAMP reserves a *memory arena*, 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.

Applications use the RAMP API, as shown in Figure 3.3, to allocate and deallocate memory segments within the RAMP arena. At startup, a configuration file is used to

```
//allocation
void* Allocate(size_t size, int segment_id);
int Deallocate(int64_t segment_id)

//migration initiation
int Connect(int process_id, int segment_id);

//transfer of ownership
int Transfer(int segment_id);
void* Receive(int & segment_id, bool paging, bool prefetch, size_t page_size);

//transfer of data
int Pull(void* address, size_t size, size_t page_size);

//migration termination
int Close(int64_t segment_id)
```

Figure 3.3: RAMP API

assign a unique process id (*p_id*) to each server and the application assigns a unique segment id (*s_id*) to each memory segment on allocation. Applications use the *p_id* and *s_id* to uniquely identify the servers and memory segments throughout the cluster. *s_ids* are used for allocation, migration, deallocation, and fault tolerance. Each coordinated memory segment occupies a fixed-length contiguous range of virtual addresses within the RAMP arena.

RAMP uses Zookeeper to track allocated and free space within the RAMP arena. By coordinating through Zookeeper, the system ensures that a memory segment allocated by one process does not overlap memory segments allocated by any other RAMP process. Allocating a coordinated memory segment in a process P_i causes that associated virtual address range to be mapped and available to process P_i, but not to any other RAMP processes. This reservation across processes, as shown in Figure 3.2, allows RAMP to avoid serialization and deserialization costs when migrating memory segments. RAMP assumes that servers have the same memory architecture. Once a memory segment has been allocated, the process P_i is free to read and write to virtual addresses within the memory segment and RAMP considers process P_i to *own* the memory segment. Further details of allocation and deallocation are discussed in Section 4.3. Once a memory segment has been allocated, RAMP does not interfere with memory access by the process that owns the segment. All reads and writes are performed directly in local memory.

3.4 Migration

The key functionality of RAMP is the ability to migrate a memory segment from one RAMP server to another. Ideally, migration in RAMP would *atomically* and *instantaneously* transfer a segment from a source process to a destination process on another server. Specifically, the source would be able to read and write to the segment at local memory speeds up until migration. After this migration, the source would lose its ability to access the segment and the destination would gain read and write access to the segment in its local memory. The destination would be able to read the final writes of the source as of the migration point.

In practice, however, this is not possible. Segment migration in RAMP diverges from the ideal case in two ways. First, migrations are not instantaneous. There is a period during which neither the source nor the target can access the segment. RAMP's migration procedure is intended to keep this window as short as possible (a few microseconds), regardless of the size of the segment being transferred. Second, the destination process may not be able to access the segment at local memory speeds for some window of time immediately after the migration. RAMP's design minimizes this access penalty while keeping the window short.

To achieve such migrations, RAMP decouples the transfer of ownership from the movement of data during segment migration. The ownership of the segment is transferred eagerly. The segment data is then pulled from the source by the destination, either on-demand or asynchronously, in small chunks after ownership transfer. This general approach of separating ownership transfer from data movement has also been used in other systems [16, 18, 31, 50]. Separating ownership transfer from data movement keeps the former operation short while migrating data gradually helps keep the access penalty low.

Section 3.4.1 describes the ownership transfer operation and the Section 3.4.2 presents data pulling mechanisms in detail.

3.4.1 Ownership Transfer

Figure 3.4 illustrates RAMP's ownership transfer protocol. The transfer is initiated by the application when the source calls `Connect` (point 1 in figure), specifying the memory segment (S) to be transferred and the destination process. On `Connect`, RAMP establishes a reliable RDMA connection from the source to the destination and registers S with the RDMA NIC at the source (pinning S in physical memory). This will later permit one-sided RDMA access to S from the destination. The source then sends (via the RDMA

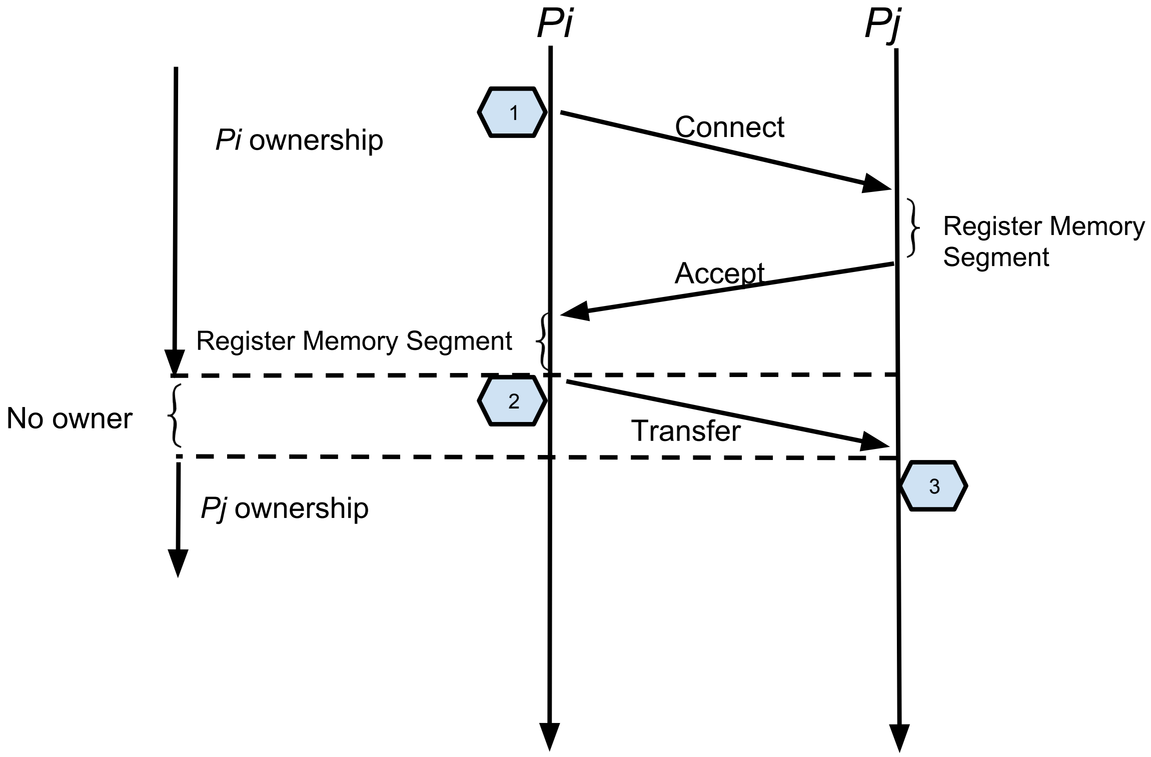


Figure 3.4: Transfer of Ownership in RAMP

connection) a *Connect* message to the destination specifying the starting address and size of S. At the destination, RAMP maps the segment's virtual addresses and registers it with the RDMA NIC. The mapping is possible because virtual addresses associated with this segment were reserved in the RAMP arena at allocation. Once this operation succeeds, the destination process is ready to receive the incoming segment S and responds by sending an *Accept* message back to the source process indicating that the connection setup was successful and the transfer operation can be executed.

This connection process is relatively expensive as it involves RDMA connection setup, a two-sided RDMA message exchange, and RDMA memory registrations (see Section 2.3 regarding the cost for memory registration) on both sides of the migration setup. However, the source retains ownership of S during the entire duration of this process and can continue to read and write to the segment. The connection setup introduces a delay from the point at which the source decides to migrate S to the actual ownership transfer but this lag does not impact memory access latency at the source.

The actual transfer of ownership does not occur until the source calls `Transfer` (point 2 in figure) and the destination calls `Receive` (point 3 in figure). When the source calls `Transfer`, it gives up its right to read or write to S. RAMP then sends a *Transfer* message to the destination process notifying it that the ownership has been transferred. The application at the destination then receives the address and s_id of the segment on the `Receive` call. The destination process may execute read and write operations in S even though it does not yet have the relevant segment state.

To ensure that these properties are upheld, RAMP must regulate access to the memory segment. However one of the RAMP design goals is to not interfere with local read and write operations of a process. RAMP enforces single reader and single writer semantics by managing memory protection using *mprotect*. As the segment ownership is being transferred RAMP `mprotects` the virtual addresses at the source process, thus preventing any local reads and writes. This ensures that state of the memory segment is consistent at the source process and that the destination process can read the last writes at the source. At the destination, the same virtual addresses are already protected because they are within the RAMP arena. As the destination receives ownership, RAMP updates these settings to allow the process to have read and write access.

3.4.2 Data Pulls

A successful transfer operation does *not* migrate the data in a memory segment. Instead, the source retains the data and the transfer mechanism only makes the data available remotely to the destination. The destination must pull the data using the `Pull` method in the RAMP API. The data pulls can either be managed explicitly by the application or the application can use the Paging layer to automate data migration.

When the destination calls `Receive`, it sets the modes of data migration. RAMP supports *Pulls*, *Paging*, and *Prefetching*. The `Pull` operation reads a contiguous chunk of memory within the memory segment specified by the application. Without paging and prefetching, the application controls all reads and writes to the memory segment, but it needs to `Pull` the relevant state before performing any operations. With paging enabled, RAMP regulates access to a memory segment by `mprotecting` the segment's virtual addresses, thus preventing any reads or writes. The data is then pulled in small chunks, i.e. *pages*, as the application attempts to access the segment. With prefetching, RAMP instantiates a thread to automatically pull in all virtual addresses of the memory segment in addition to the paging mechanism. The pull, paging, and prefetching mechanisms can be used together on the same process. Section 3.4.3 provides further details about paging and prefetching in RAMP.

When all the relevant data is pulled by the application, the destination process uses `Close` to terminate the migration. `Close` serves two purposes:

1. It cleans up the migration resources
   - S is de-registered at the source and destination.
   - S is unmapped at the source.
   - The RDMA connection is shut down and associated objects are freed.
2. RAMP requires the destination to terminate the migration of S using the `Close` operation before it can initiate a new migration of S to another destination. Note that it is not necessary for the application to pull all of S before closing the connection. This permits applications to only pull parts of S that are required and in use.

3.4.3 Paging Layer

The paging layer automates data migration. When a memory segment is received with paging enabled, RAMP regulates application access to the memory segment by using `mprotect`. The application does not need to synchronously pull sections of a memory segment before accessing them. Instead, it accesses data as if it was locally available. On a memory access, the protection settings raise a SIGSEGV fault. RAMP then intercepts the SIGSEGV notification, pulls in the relevant virtual addresses and updates the protection settings to allow local read and writes. The control is then returned back to the application process which then executes the last instruction and continues processing, oblivious to the remote fetching event. This process is called *demand paging*. In addition to demand paging, applications can enable *prefetching* as well. With prefetching, RAMP initializes another thread which automatically pulls in pages of S and updates the paging protection settings until all virtual address of S have been copied. The prefetching mechanism works in tandem with demand faults and pulls in one page at a time to keep access latency low. When prefetching is enabled, the `Close` operation blocks until RAMP copies the entire memory segment.

Both paging and prefetching can be used alongside explicit pulls directed by the application. RAMP tracks the state of each page to ensure that the data is pulled only once. Internally, RAMP tracks pages in units of 2 KB and applications are restricted to issue pull requests in multiples of the base unit. The base unit of 2 KB is experimentally evaluated to ensure maximum latency gain with RDMA RC. All pages are initialized to a default state of *Remote* when a `Receive` is called. They are atomically updated to *In-Flight* state when a request is issued on them. Finally, the state is set to *Local* after the protection settings have been updated to permit local read and write operations. A `Pull` request is only issued on a page if an atomic-compare-and-swap operation successfully updates the state from *Remote* to *In-Flight*. The RAMP exception handler can also differentiate between SIGSEGV signals generated by read and write operations within a protected memory segment from illegal memory accesses outside all protected memory segments. RAMP searches the virtual address that generated the SIGSEGV in the list of memory segments that have been protected. If the virtual address originates from within the protected regions, RAMP handles the signal by fetching the remote pages or restarting the operation if the pages are already in-flight, while the remaining signals are passed onto the application.

The paging layer provides flexible data migration. As memory segments grow larger, a single `Pull` to migrate the segment data (*stop-and-copy*) will take more time. Instead of pulling in the entire memory segment with a stop-and-copy operation, the application can enable paging and use `Pull` as a form of directed prefetching. This allows S to be available to the destination process throughout the migration. Furthermore, RAMP's prefetching can be used if the entire memory segment needs to be migrated. The paging layer allows applications to determine the best migration mechanism and control the trade-off between access latency and bandwidth utilization

3.5 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 memory segment. There is *no need to serialize or deserialize the structure* when it migrates since a memory segment's position in the virtual address space of the destination is the same as its position at the source. Migratable containers allow applications to operate at a higher level of abstraction, at which data structures, rather than memory segments, are migrated.

To illustrate this usage pattern, RAMP includes *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 that ensures that all memory allocated for the container and its contents lies within a segment. The underlying memory segment is allocated from within the RAMP arena when the container is first created. The scoped allocator is created within the memory segment as well. Second, the container in RAMP is equipped with an additional constructor, which is used to initialize the container at the destination after migration. Finally, the container's normal interface is supplemented with additional methods, analogous to those in Figure 3.3, to provide container migration capability. Apart from the migration capability, 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 transfer the container's underlying memory segment. At the destination, the application immediately begins using the container as soon as it receives ownership of the segment. RAMP's paging is used to migrate the segment data on demand as the application uses the migrated container at the destination. In addition, the application can use RAMP's explicit paging mechanism to pull (prefetch) chunks of segment data from the source. The scoped allocator tracks parts of the memory segment that have been allocated. Using this information, the container `Pulls` only the allocated parts of the memory segment from the source.

With the exception of the custom memory allocation and the additional migration related methods added to containers in RAMP, it is *not* necessary to modify the interface or implementation of the underlying STL container to support migration.

Chapter 4

Fault Tolerance

RAMP is a migration platform for distributed applications running on a cluster of machines, where arbitrary server failures can result in the loss of data residing in a memory segment or metadata such as the allocation state of the RAMP arena or segment ownership information. RAMP does not protect against the loss of data within a memory segment due to server failures. This functionality was omitted to keep RAMP's design simple and lightweight. Also, since many distributed applications already use logging or replication to protect data integrity, building data protection functionality in RAMP would be redundant and complicate its design.

In RAMP, if a process p_i fails, the contents of the memory segments it owns will be lost. In addition, for any ongoing migration of a memory segment S from process p_i to process p_j for which ownership has been transferred but the migration has not been completed, a failure of p_i will result in loss of data that had not been migrated. For such cases, data pulls or memory segment accesses fail and RAMP raises an exception.

Although RAMP does not protect the contents of the memory segments in the presence of server failures, it is designed to preserve the allocation state of the RAMP arena and segment ownership information. This functionality is critical for RAMP so that failures do not result in memory leaks in the RAMP arena.

The remainder of this chapter describes the failure model for RAMP, the fault tolerance properties that RAMP guarantees and how these properties are incorporated in RAMP.

4.1 Failure Model

RAMP assumes a fail-stop model for server and process failures in an asynchronous network environment with arbitrary delays. In this model, servers can crash arbitrarily and other *functioning servers* detect that a server has crashed. On a failure, servers stops responding to requests. RAMP uses Reliable Connection (RC) RDMA connections for messages. RC is comparable to TCP and guarantees the following:

1. All packets/messages are reliably delivered and acknowledged to the sender.
2. All packets/messages are delivered in the order in which they were sent.

Therefore, RAMP assumes that all acknowledged messages in RAMP have been delivered.

4.2 Properties

The goal of fault tolerance in RAMP is to preserve the allocation state and reliably track ownership information for allocated memory segments. As such, RAMP guarantees the following properties:

1. At any given point, a memory segment will have at most one owner process.
2. Allocation state of the RAMP arena and memory segment ownership information are persistent.
3. Allocation and deallocation operations in RAMP are atomic, i.e., once these operations are issued they will either succeed entirely or fail entirely.
4. Transfer operations may fail as a result of a failure of the source or the destination process, leaving either the source, or the destination, or neither (but not both) with the ownership of the segment.
5. Applications can query memory segment ownership information for failed RAMP processes. RAMP will identify all memory segments owned by the failed process, as well as any segments whose ownership is in doubt due to a failed transfer to or from the specified process.

The last property enables the application to clean up resources after process failures by querying RAMP for the memory segments owned by the failed process and then deallocating them. The deallocation, in turn, enables RAMP to clean up the allocation state of the RAMP arena to prevent memory leaks.

4.3 Implementation

To ensure the properties described in Section 4.2, RAMP uses Zookeeper to track and maintain the global state of allocations and memory segment ownership. Zookeeper ensures that the global state is replicated and persistent. We extend the RAMP API (Figure 3.3) with a single method, `get_segments`:

```
vector<segment_info> get_segments(int process_id);
```

`get_segments` allows any process to query the memory segments owned by a given process. It is described in more detail in Section 4.3.3.

4.3.1 The Global State

RAMP stores and persists the global allocation and memory segment state for RAMP using Zookeeper. The allocation and deallocation state of the RAMP arena is stored in an *allocation node* while the memory segment state is stored in *memory segment nodes* in Zookeeper's hierarchical namespace, as shown in Figure 4.1.

The allocation node consists of a memory allocation list and a memory free list serialized as strings within a single Zookeeper node. These lists store information for all virtual addresses allocated and deallocated in the RAMP arena. For allocation and deallocation, RAMP reads the latest state of the allocation node and updates it atomically. Further details for allocation and deallocation operations are discussed in Section 4.3.2.

As memory segments are allocated, RAMP creates a memory segment node to track the ownership information (and other metadata) for the memory segment. These nodes are stored as children of the `memory-segment-info` node using the segment id specified by the clients at allocation. RAMP ensures that segment ids are unique. For any given memory segment S, RAMP stores the segment_id, source process id, destination process id, the starting virtual address in the RAMP arena, and the size of the memory segment. The source process id is always set and represents the process that owns the memory segment. The destination process id is only set when the ownership of a memory segment is being transferred from the source process to the destination process. During this interval, the ownership of a memory segment is *in doubt* and the global state cannot determine which process owns the memory segment. RAMP's ownership transfer protocol, described in Section 4.3.2, is designed to keep this window short. RAMP uses the global state to answer the `get_segments` queries and determine which memory segments were owned by a process.

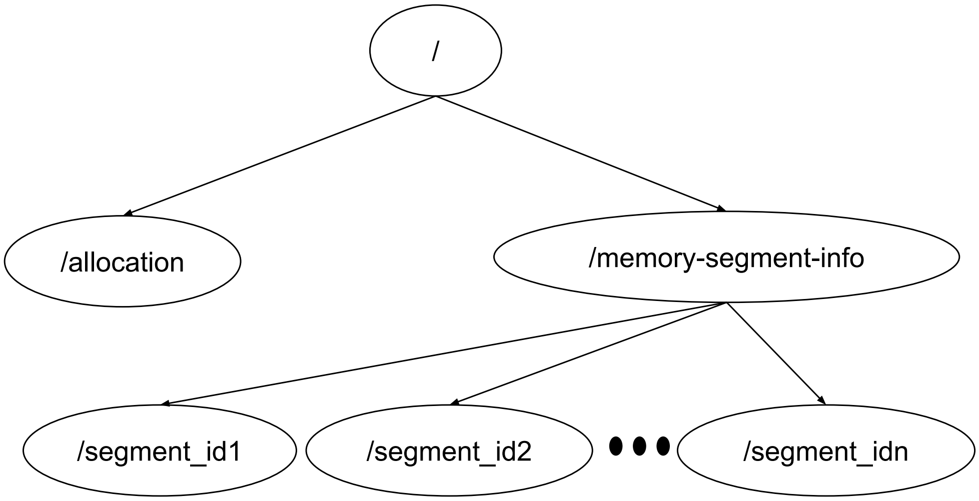


Figure 4.1: RAMP Structures in *Zookeeper*

4.3.2 Operations

This section provides details describes how RAMP operations store and maintain global state.

Allocation and deallocation in RAMP are coordinated via Zookeeper using the allocation node. For an allocation at process p_i, RAMP reads the allocation state from Zookeeper and prepares a synchronous and atomic *multi* operation to update the allocation node and create a memory segment node with the source process set to p_i. The multi operation in Zookeeper is a set of operations that are guaranteed to execute as a transaction. For deallocation, RAMP reads the allocation node and prepares an atomic multi operation to update the free-list and delete the memory segment node created to track the ownership information for the memory segment. RAMP relies on Zookeeper's optimistic concurrency control mechanisms to ensure that updates to the allocation node are atomic. Both allocation and deallocation operations in RAMP are idempotent, as long as the application does not recycle the segment ids. When there is high contention on the allocation node, the multi operation may fail. For such cases, RAMP restarts the allocation or deallocation operation.

Migrations in RAMP consist of an ownership transfer with the operations `Connect`, `Transfer`, `Receive`, and `Close` and data transfers with the `Pull` operation. Figure 4.2 shows how RAMP's ownership transfer mechanism is extended to provide fault tolerance. The figure illustrates the transfer of a segment S from a source process (Src) to a destination process (Dst) with RAMP's fault tolerance mechanisms. The protocol ensures that the global state is updated with atomic writes to Zookeeper and that these updates are not on the critical path of an ownership transfer, i.e., the `Transfer` operation. Operation 1 synchronously adds the destination id to the memory segment node with an atomic Zookeeper write operation once the `Connect` operation is completed and before the *Transfer* message is sent. The memory segment ownership is now 'in doubt', as the global state has both the source and the destination set. Operation 2 asynchronously updates the memory segment node with an atomic Zookeeper write operation and sets the source process id to the destination process id and clearing the destination process id after the destination process receives the `Transfer` message from the source. The memory segment ownership is now known (as Dst). While the `Transfer` message is being sent, neither the source nor the destination process can access the memory segment. RAMP's design ensures that updates to the global state do not increase this period of inaccessibility. RAMP does not guarantee progress of migration operations. However, RAMP uses RDMA connection timeouts to maintain the open connections until they are either closed by the application or broken due to network failure.

Data transfer operations do not update the global state. However, until all of the data in a memory segment has been pulled, a failure of the source server may cause `Pull` operations to fail at the destination server. Pull failures notify the application that the remote virtual addresses of the memory segment are no longer available. In addition, RAMP also de-registers the memory segment and cleans the connection object when an RDMA connection breaks or times out. This operation does not affect RAMP's fault tolerance guarantees but does allow RAMP to free resources that were allocated for the migration.

4.3.3 get_segments

The get_segments call can be used by the application to get a list of memory segments that were owned by a process that has failed. RAMP tracks the memory segment ownership globally as described in Section 4.3.2. On a call to get_segments for process p_i, RAMP reads the latest snapshot of the global state for memory segments stored at Zookeeper. RAMP then searches the memory segment nodes and returns the segment ids from all of the nodes for which the source or the destination is p_i. This list represents the memory segments that might have been owned by p_i. To prevent memory leaks in the RAMP arena, a safe action for the application is to deallocate all such memory segments.

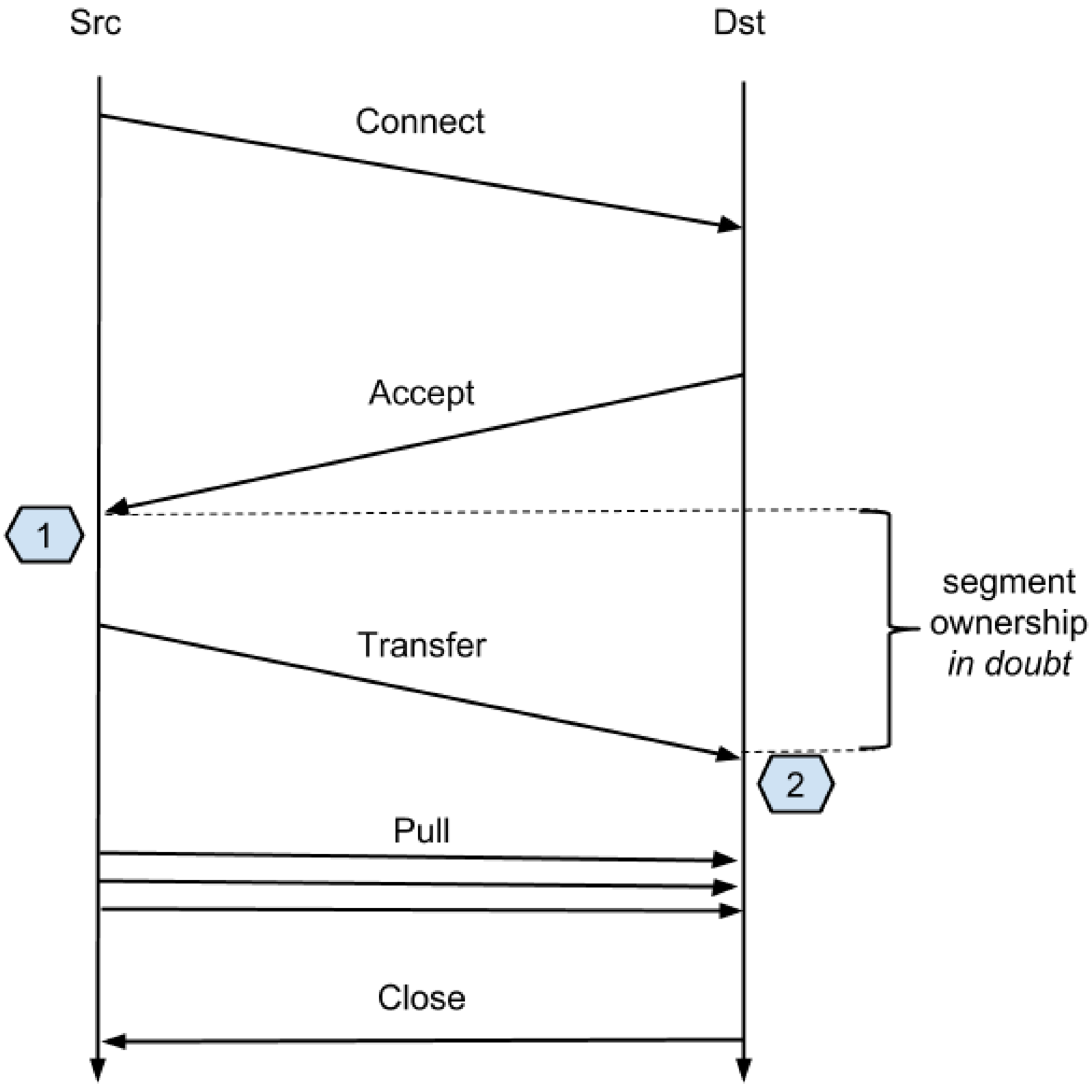


Figure 4.2: Ownership Transfer with Fault Tolerance
1. *Src* sets *Dst* as destination synchronously in Zookeeper
2. *Dst* adds *Dst* as source and clears destination asynchronously in Zookeeper

Chapter 5

Evaluation

This chapter presents an empirical evaluation of the RAMP platform. We use the following sets of experiments to evaluate RAMP:

1. Microbenchmarks to evaluate RAMP ownership transfer and data migration operations.

2. A case study of a loosely coupled application that uses RAMP to perform live load balancing.

Note that the experiments do not use the fault tolerance features in RAMP.

All experiments are conducted on a cluster of Supermicro SYS-6017R-TDF servers with one Mellanox 10GbE SFP port, 64 GB of RAM, and two Intel E5-2620v2 CPUs each having 6 cores with a frequency of 2.6 GHz. Each node is connected to a Mellanox SX1012 10/40 GbE switch. All nodes run an Ubuntu 14.04.1 server distribution with Linux kernel version 3.13.0.

5.1 Ownership Transfer

In order to evaluate ownership transfer, we set up two experiments. The first evaluates the time required for complete ownership transfer of a memory segment from a source node to a destination node, as described in Section 3.4.1. This includes the time to set up and close the transfer, as well as the actual transfer itself. The second experiment focuses on the latency of the actual Transfer operation and the *Transfer* message sent as part of this operation. The *Transfer* message represents the critical period for an ownership transfer, since the segment has no owner while the transfer occurs. Both experiments are repeated for memory segments of sizes ranging from 1 MB to 512 MB.

For the first experiment, we evaluate the total time it takes to migrate ownership of a memory segment from a source process to a destination process. We set the source process on one server and the destination process on another server. The source process allocates a memory segment and then calls Connect and Transfer and the destination process calls Receive and Close without pulling in any data. This ensures that data transfers do not add additional latency to the ownership transfer process. The memory segment is then deallocated at the destination process. At the source, we measure the total time taken from Connect to receipt of the Close notification from the destination. The experiment is repeated 10 times for a memory segment of a given size. We plot the mean latency over the ten runs as well as the standard deviation.

Figure 5.1 shows the results as a function of the segment size (note that standard deviation is too small to be visible in the figure). Although no data is transferred the migration latency increases linearly with the size of the memory segment. We experimentally determined that this latency is dominated by the time required to pin the virtual memory during the RDMA registrations which occur as a result of the Connect call. Moreover, the total latency for migrations is within a few milliseconds ($\sim$2 ms) of twice the memory registration time of a memory segment (see Figure 2.5). This is expected, as the Connect call registers the memory segment twice, once at the source and once at the destination.

Although it can take hundreds of milliseconds to transfer the segment ownership, the segment remains accessible to the source node for most of this time. There is only a short window of unavailability after Transfer is called at the source and before the Receive returns at the destination. We set up a second experiment to measure this window. It is difficult to measure this window as it starts at the source process, p_i, on one server and ends at the destination process, p_j, on another server. Therefore, in order to correctly capture this duration, we set up an experiment that synchronizes two migrations such that we can measure, at the source process, the time taken for two transfers. The experiment is set up as follows: p_i allocates a memory segment S_i and p_j allocates another memory segment S_j of the same size. p_i connects to p_j and vice versa. However, the memory segments are not immediately transferred. After both S_i and S_j have Connected (i.e. registrations for both segments have been completed), p_i begins the experiment by starting a clock and calling Transfer on S_i to p_j. On Receive of S_i at p_j, p_j calls Transfer on S_j to p_i. p_i stops the clock on Receive of S_j. This records twice the time taken to transfer a memory segment. Half of this window of time represents the time taken for a Transfer operation,

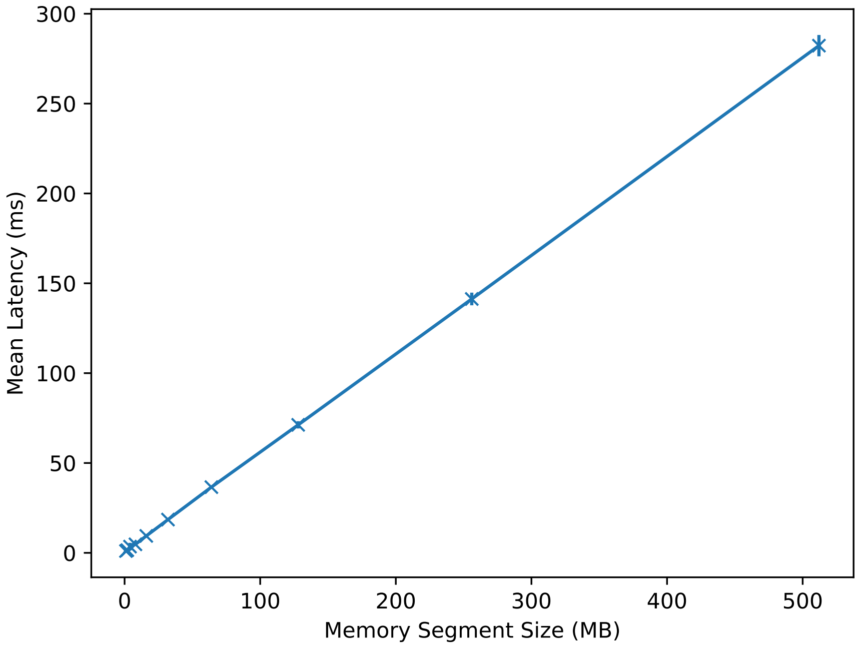


Figure 5.1: Latency of Ownership Transfer

it captures the latency of the RDMA transfer message and the processing time in RAMP. In an effort to better understand the transfer operation we also record the round-trip time for the RDMA transfer message from p_i to p_j on `Transfer` using the RDMA notification sent back to p_i on a successful delivery to p_j. The experiment is repeated 10 times for a memory segment of a particular size and the results are halved to show the latency for a single `Transfer` operation and one side of the RDMA transfer message. We plot the mean and standard deviation of these times across all 10 runs.

Figure 5.2 shows the results of this experiment. The duration of the critical transfer operation is about 40-60 μs, independent of the size of the memory segment. The RDMA transfer message has a mean latency of around 15-20 μs which indicates that RAMP processing takes up a majority of the time during the transfer operation. There was some variation for each run in this experiment, although the long error bars in Figure 5.2 are due to 1 or 2 outliers in each run.

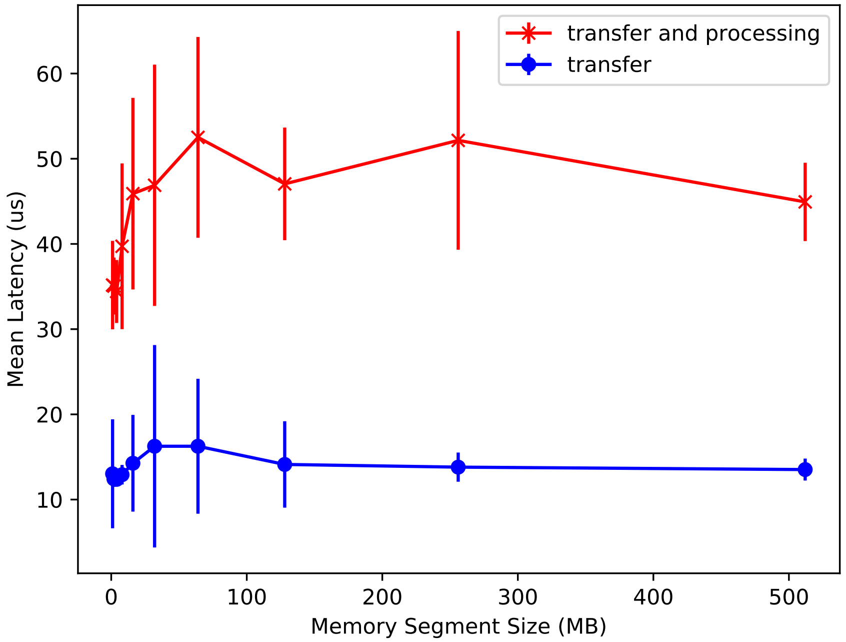


Figure 5.2: Latency of the `Transfer` Operation

5.2 Data Pulls

Migrations in RAMP have two parts, an ownership transfer and data pulls. The ownership transfer does not itself migrate any data. Data is moved using subsequent `Pull` calls. A targeted use case for RAMP is one in which the application at the destination pulls data from the source on demand as it accesses the migrated segment. We set up experiments to measure the performance impact of these data pulls on the application at the destination.

We set up two experiments, each with a C++ unordered map contained within a memory segment. In the first experiment, the segment size was 128 MB and in the second experiment the memory segment size was 256 MB. Each map was populated with pairs of 8 byte keys and 128 byte values until the entire memory segment was utilized by the container (approximately 500 thousand and 1 million entries, respectively). To evaluate data pulls, each segment was migrated from a source node to the destination node using RAMP's ownership transfer mechanism. At the destination, a single application thread begins performing *get* and *set* operations (1:1 ratio) on the unordered map in a tight loop immediately after receiving the incoming container. We ran three different versions of the experiment to test different ways of pulling segment data. In the first version, the container data is *pulled on demand*, with a page size of 4 KB. In the second version, demand faults are supplemented by sequential *prefetching*. In the third, which we refer to as *stop-and-copy*, the application uses a single large `Pull` request to pull all of its data before allowing any container operations.

Figures 5.3 and 5.4 illustrate the latency of container operations as a function of time for containers of sizes 128MB and 256 MB respectively. Time zero represents the point at which the destination node receives ownership of the container. We show the mean and 95th percentile latency, measured over 100 ms windows.

By using the stop-and-copy approach, all container accesses are performed at local memory speeds, with tail latencies of 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 is pulled. This unavailability window grows from 230 ms to 350 ms as we increase the memory segment size from 128 MB to 256 MB. This unavailability window depends on container size and link bandwidth. With paging, the container is available immediately but there is a period of about 0.75 s (128 MB) and 1.5 s (256 MB) during which the container access latencies are elevated. Tail latencies remain below 50 μs during most of this period. Prefetching pulls the container data even faster, with the container access latencies dropping to local speeds after around 0.65 s (128 MB) and 1.3 s (256 MB). Prefetching also lowers the average access latency while the data is pulled but results in higher latencies immediately after migration. This is because RAMP uses a single RDMA connection between the source and the destination, causing prefetches to delay demand pulls. The initial higher latency can be mitigated by establishing a second connection for prefetches. 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. Tail latencies are unaffected by the size of the container.

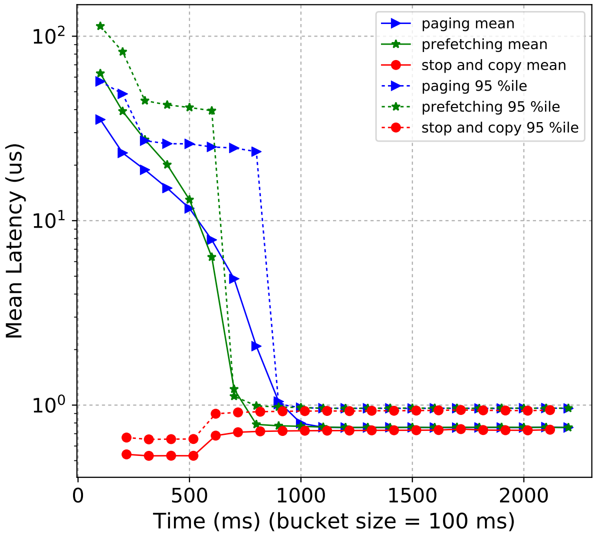


Figure 5.3: Post-Migration Container Access Latency (Size = 128 MB)

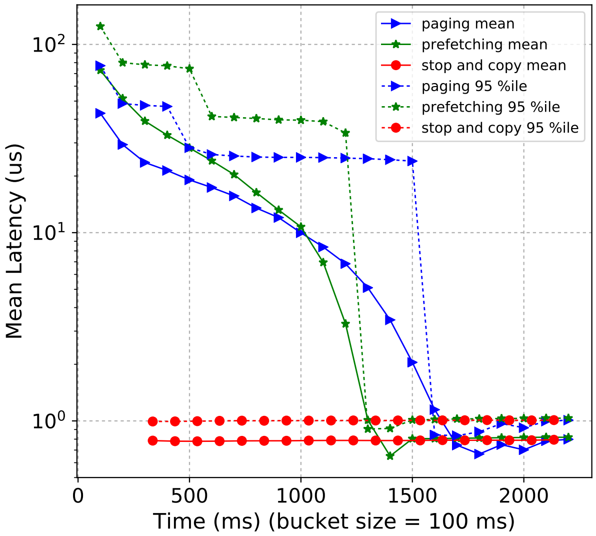


Figure 5.4: Post-Migration Container Access Latency (Size = 256 MB)

5.3 Live Load Balancing Case Study

RAMP is designed to provide coordination services for applications to reconfigure data within a cluster. In order to illustrate the use of RAMP for migrations, we conducted a case study in which we use RAMP for load balancing operations in a loosely coupled distributed application under load.

For the purposes of this experiment, we built a loosely coupled distributed application called ***rcached***, based on *memcached* [19]. **memcached** is a widely used distributed in-memory key-value storage system. rcached is a simple drop-in replacement for memcached, with the added ability to perform load balancing operations using RAMP. rcached uses hashing to partition the key space, and stores each partition in a C++ unordered map contained within a RAMP memory segment. rcached uses consistent hashing to map keys to partitions and subsequently partitions to the backend servers. Each server is responsible for a subset of the partitions. rcached uses a consistent hashing ring to migrate partitions and it enables paging on Receive. rcached servers currently do not use prefetching.

When a partition migrates, rcached clients receive a negative acknowledgement from the source server as they attempt to access the partition. Clients then redirect their requests to a new server which is determined by the consistent hashing ring. Clients continue to redirect requests to the server next on the list until the partition is located. No data is lost during rcached migrations. While rcached is not as heavily engineered as memcached, 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.

The load balancing rcached experiment was configured with a cluster of four rcached servers which store 40 million keys partitioned in 128 partitions (c++ unordered maps). 100 closed-loop clients issue *get* requests, using a Zipf distribution with parameter 0.99 over the key space and no think time. Figure 5.5 shows the client side request latencies averaged over windows of 40,000 requests and broken down by server id. The results show that this configuration has some request skew as the load is not perfectly distributed across all servers. Server 1 ends up receiving the most requests and as a result experiences higher latency. Overall, the average request throughput was about 380,000 requests per second over the experiment.

To test migrations, we ran the same experiment again. This time after an initial warm up period, we used RAMP to migrate two heavily loaded partitions from server 1 to server 2 to better balance the load. Figure 5.6 shows the results of the load balancing experiment.

We report the mean client-side request latency averaged over windows of 40,000 requests , broken down by the server that handled the request. The two vertical lines indicate the approximate time at which a partition migration occurred. The overloaded server 1 experiences a drop in load while the all other servers including server 2 experience an increased load. This is because of the closed-loop nature of our clients. As expected, Server 2 also shows short spikes in mean and 95th percentile (not shown) at the time of migrations. However, these spikes are no worse than regular `rcached` latency variation due to short-term load fluctuations. The overall throughout after migrations increased from 380,000 requests per second to 390,000 requests per second.

This case study shows that RAMP can successfully balance load for a loosely coupled distributed application with minimal interruptions. RAMP behaves as expected; ownership transfers do not significantly impact the availability of partitions in `rcached`. Also, the data pulls have minimal effect on client-side latency and the application achieves lower latency per server as a result of reconfigurations with RAMP.

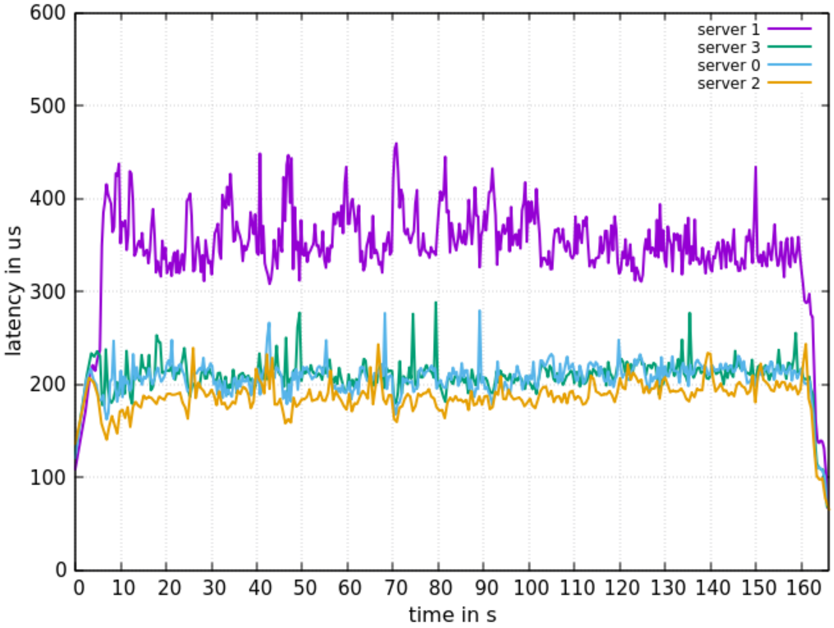


Figure 5.5: Rcached Without Load Balancing

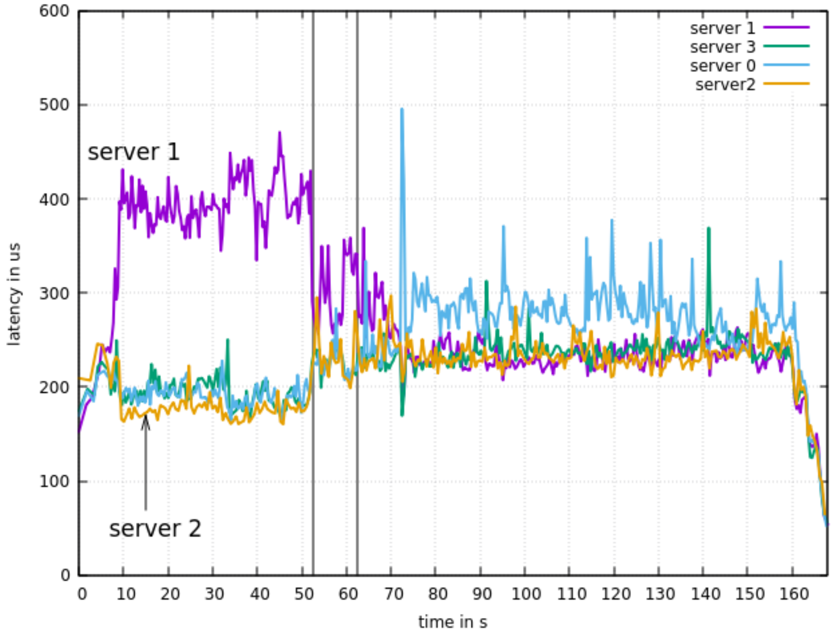


Figure 5.6: Rcached With Load Balancing

Chapter 6

Related Work

RAMP is related to work in three different research areas: distributed shared memory systems, RDMA-based systems, and migration platforms. In this chapter, we survey the previous work done in these areas.

6.1 Distributed Shared Memory Systems

Protic et al. [47] surveyed the Distributed Shared Memory (DSM) landscape in 1996 and extensively covered DSM algorithms, systems, and consistency models. DSM systems offer a general shared memory programming model, using either implicit message passing or an extended cache coherence protocol for private (local) and shared (remote) memory. DSM algorithms deal with data distribution and accesses by leveraging compiler and operating system support for data invalidation, access, and update protocols. Ivy [41], Blizzard [36], and Mermaid [55] are examples of popular DSM systems that do not use RDMA.

Several software-based DSM systems use RDMA [42, 43]. Noronha et al [42] present NGDSM, a software-based cache coherence protocol using RDMA primitives. The same authors also explore using RDMA for reducing diff overheads for software based DSM systems. These systems are similar to shared memory systems discussed in Section 6.2.

RAMP has similar properties to a DSM system as it allows processes to share data in certain sections of its virtual address space. However, RAMP is not a DSM system as it does not try to invalidate data access or use data update protocols. Instead, RAMP is designed to cater to applications that migrate large amounts of data, while minimizing access penalties. Unlike DSM systems, memory segment ownership and data movement in RAMP is controlled by the application.

6.2 RDMA-Based Systems

There has been significant research and industrial interest [44, 48, 33] in RDMA-based systems in last few years. These systems can broadly be categorized as offering either a *shared-storage* abstraction or a *shared-nothing* abstraction. Systems offering a shared memory abstraction use cluster-wide memory to expose a shared address space (like a DSM) or a database. Systems that offer a shared-nothing abstraction use RDMA to provide a message passing model among separate per-server address spaces. In the following sections, we present both types of RDMA systems.

6.2.1 Shared Memory Systems

FaRM [14, 15] exposes the memory of a cluster of nodes as a shared address space, leveraging RDMA to build a computing platform. FaRM offers lock-free reads and transactional support with strict serializability, using two-phase commit (2PC) across the cluster to commit transactions. Several other systems [40, 26, 8, 57] leverage RDMA and other hardware features, such as hardware transactional memory (HTM), to build low latency and high throughput transaction processing systems or key-value stores. These systems also expose the memory of a cluster in a similar fashion as FaRM and employ various concurrency control mechanisms.

Nam-DB [59, 7] and Tell [34] are RDMA-based database systems that offer snapshot isolation guarantees. Nam-DB uses one-sided RDMA reads to remotely read data items for each access. Nam-DB uses global counters to track the latest snapshot of a data item and two-sided RDMA messages for index lookups. Nam-DB foregos the inherent message ordering and delivery guarantees of RDMA. On the other hand, Tell decouples transactional query processing from data storage by using a centralized commit manager and distributed multi-versioned concurrency control (MVCC), using a BwTree. Tell is limited by its centralized commit manager, which is a single point of contention.

Infiniswap [24] provides a remote memory paging system for a cluster of machines. It handles memory overflows on a machine by using the un-utilized memory on the other machines in the cluster. It allows memory disaggregation in a cluster, where underutilized memory on one machine can be utilized by another machine. By using Infiniswap, processes can avoid the disk access penalty due to pages being swapped in and out of disk.

In all these systems, access to data in shared memory may require access to a remote server. These accesses require coordination, which requires some kind of locking, eviction, or versioning strategy. In contrast, RAMP's programming model provides single reader, single writer semantics, which removes the need for regulating concurrent accesses. Another notable difference is that applications may need to pay network costs each time they want to access it. RAMP ensures that all memory accesses are local except during application-initiated migrations.

6.2.2 Shared-Nothing Systems

FaSST [27] is an in-memory transaction system that focuses on scalability by building an RPC abstraction on top of one-to-many unreliable datagrams using the SEND and RECEIVE RDMA verbs. FaSST cites the storage limitation of current RNICs and the amount of stored state required by each reliable persistent connection for their design choice. FaSST shows that the limited storage available on RNICs limit the scalability of its design. Su et al. [53] present another way to build an RPC abstraction over RDMA. They argue that one-sided RDMA reads and writes do not maximize the RNICs capabilities due to the asymmetry between incoming and outgoing message rates. They further argue that performance degrades as systems need to regulate access to shared memory with some kind of locking or versioning scheme. They build and evaluate a remote procedure call (RPC) system based on one-sided RDMA message transfers. Clients write to remote server memory to send requests, and then read responses from remote server memory on completion of the request.

DARE [46] and APUS [54] are consensus algorithms based on the RDMA networking stack. The idea behind these systems is to improve the runtime, throughput and scalability of state machine replication algorithms by exchanging small messages between different servers quickly. These systems share memory specifically to improve message passing and coordination between the consensus groups leveraging single (DARE) and multiple (APUS) connections.

Other research [6, 21, 49] focuses on distributed join processing with RDMA. Barthels et al. [6] primarily use pinned buffers to accelerate the data flow between servers, each of which is computing a part of the join. Frey et al. [21] suggest a new architecture for transferring messages between participating nodes and evaluate their approach for different distributed join processing algorithms. Flow-join [49] suggests a lightweight adaptive skew handling technique for distributed join processing systems and exchanges messages so that skew in workload does not impact performance.

Shared-nothing systems use RDMA to take advantage of the performance gains of new hardware technologies. However, they do not fully utilize RDMA's performance benefits as data still needs to be serialized for transfers and deserialized on receipt. Furthermore, most shared-nothing systems can only support small message transfers and their performance degrades for large message transfers. On the other hand, RAMP is designed to fully utilize the RDMA's performance benefits and supports transfers of large memory segments and structured data without serializing and deserializing across servers with the same memory architecture. RAMP also avoids the limitations of RNICs asymmetry for incoming and outgoing messages, conflict resolution mechanisms used by shared memory abstraction systems, and RNICs limitations in storing states by only establishing connections on demand and enforcing single writer semantics.

6.3 Data Migration Techniques

The key functionality of RAMP is the ability to migrate potentially large memory segments between servers without disrupting access. Most distributed systems, especially distributed database management systems (DBMS), face a similar issue where data movement leads to performance degradation. As such, data migration techniques have been widely studied. In this section, we present various data migration techniques and how they relate to RAMP.

Stop-and-copy is the simplest migration approach. In stop-and-copy, the system stops processing requests at the source, copies relevant state to the destination, and then continues execution of the incoming requests at the destination. This approach results in significant downtime and performance degradation. Alternately, the *Synchronous migration* [17] approach avoids this downtime by creating a replica of the partition or state at the destination. Depending on the implementation, either logs or snapshots are used to bring the replica up to date with the source. Requests are then executed at both the source and the destination. After synchronization, the source may simply use a failover mechanism to make the destination the primary replica and complete a migration.

Albatross [12] is a live migration system that tries to avoid the downtime of stop-and-copy by using a *flush-and-copy* mechanism. Albatross [12] copies a snapshot of the partition asynchronously to the destination. Incoming updates are iteratively shipped to the destination until the partitions at the source and destination converge. On convergence, Albatross uses an atomic handover to move ownership of the partition to the destination.

This may cause a small downtime if there are pending updates to be shipped. *Slacker* [5] is another system that uses the live migration technique but minimizes the impact of migration by throttling the rate at which pages are migrated.

Zephyr [18] migrates partitions by allowing concurrent execution at the source and destination without using distributed transactions. Incoming requests get rerouted to the destination forcing a pull. Any requests at the source for data that has been migrated need to be restarted at the destination. Zephyr's approach does not cause a downtime but requires any indexing structure to be frozen for the duration of the migration. A copy of the database's physical structure is also copied to the destination at the start of the migration. ProRea [50] extends Zephyr's approach by migrating hot tuples to the destination at the start of migration to reduce load at the source.

Minhas et al. [38] describe another mechanism for live migration using the replication and recovery mechanism in a DBMS. Squall [16] allows for fine-grained live reconfiguration by closely keeping track of the tuples at both source and destination. The decentralized approach safely moves data across the nodes in the presence of distributed transactions by reactively pulling data when required at the destination. Squall requires a global lock to initialize the reconfiguration. Rocksteady [31] extends Squall by eagerly transferring ownership of the migrating partition from the source, like RAMP. RockSteady uses RamCloud's [45] server bypass mechanism for low latency access to migrate data (both reactively and pro-actively) and only serve requests at the destination. RockSteady ensures that it meets service level agreements by throttling the amount of work that the source is required to do to move the data. Unlike RAMP, RockSteady does not utilize one-sided RDMA reads for data migration, since a RamCloud tablet (the unit of migration) may be scattered in memory. Instead, it utilizes remote procedure calls over the RAMCloud stack.

Wei et al. [56] described a live reconfiguration approach which extends DrTM+R [57] using a pre-copy approach similar to Albatross [12]. They resolve several issues with the pre-copy approach by using DrTM+R's fault tolerance mechanisms. Using one-sided RDMA reads they transfer the difference between the source copy and destination copy. They further use log forwarding to ensure that source and destination partitions converge.

Live migrations have also been explored in the context of virtual machine (VM) cloning [32, 39] and migrations [10, 25]. Like DMBS migrations these systems use some combination of pulling data from the source, like RAMP, and pushing it from the source to accomplish migration. SnowFlock [32] pulls data from source to destination on demand, like RAMP, although it does not use RDMA. Clark et al. [10], like several other DBMS approaches [12, 56], rely on pushing data from the source before transferring control. This reduces the post-transfer performance penalty, but at the expense of delaying migration.

Kharabrov et al. [30], describe a technique that allows users to share higher order data structures by storing them in a global heap in Java. These structures can then be accessed by other nodes by transferring the raw data from the global heap space. The system is based on TCP/IP although it could be modified to leverage RDMA. However, this approach is Java specific and objects are immutable after migration.

Unlike other migration systems, RAMP's share-on-demand architecture provides a few unique properties. RAMP relieves the source node of CPU load for data movement after an ownership transfer and avoids serialization and deserialization costs. Data structures in RAMP can be edited by the owner process as soon as they receive ownership (with minimal latency) and RAMP data migrations work well for large transfers. Additionally, the RAMP programming model can make use of several migration techniques described in this section to further improve its performance.

Chapter 7

Summary and Future Work

RAMP is an RDMA-enabled migration platform that enables low impact data migrations for distributed applications. RAMP provides a useful abstraction for building coordination services for loosely coupled distributed applications. RAMP is based on a shared-on-demand architecture where servers exchange memory segments. The key feature of RAMP is the decoupling of the ownership of data from the migration of data. We present RAMP's programming model, which is based on memory segments, and the RAMP API, which includes operations for allocation, deallocation, ownership transfer, and data migration. Furthermore, we describe the fault tolerance properties of RAMP and discuss the guarantees they provide. In addition, we show that RAMP can be used to build migratable data structures that can be transferred between servers without significant performance impact.

We designed experiments to evaluate RAMP's ownership transfer and data migration operations. We show that a container data structure can be live-migrated using RAMP while keeping tail access latencies below 100 us and an ownership transfer can be accomplished with a delay of a few tens of microseconds. Furthermore, we demonstrate that RAMP can be used for balancing load in a loosely coupled distributed system called `rcached`, which is a distributed in-memory key-value cache.

Looking forward, there are a few optimizations and extensions that can be built to further improve RAMP. The performance of data migration in RAMP can be improved by spreading data pulls across multiple connections. This would enable RAMP to prioritize demand or explicit pulls while prefetching. Another extension to this work would be to utilize RAMP's API to implement low impact transfers using Google Protobuf [23], which is Google's library for serializing and deserializing structured data. Applications can rewrite their protobuf files and extend Protobufs to use RAMP to migrate data. This would allow distributed applications to leverage RAMP's low impact migration capabilities using the existing protobuf code in their applications. Finally, RAMP's data pull could be modified to instead push data and keep partitions synchronized across multiple servers. This functionality could be used to implement fault tolerance in databases that replicate data partitions across multiple instances for fast failovers.

References

[1] Infiniband Trade Association. Infiniband. http://www.infinibandta.org. [Online; accessed 3-January-2018].

[2] Infiniband Trade Association. RDMA over Converged Ethernet (RoCE). [Online; accessed 3-January-2018, Supplement to InfiniBand Architecture Specification Volume 1 Release 1.2.2 Annex A16].

[3] Open Fabrics Association. SoftiWARP. https://github.com/zrlio/softiwarp, 2010. [Online; accessed 8-March-2018].

[4] Open Fabrics Association. SoftRoCE. https://github.com/SoftRoCE, 2014. [Online; accessed 8-March-2018].

[5] Sean Barker, Yun Chi, Hyun Jin Moon, Hakan Hacigümüş, and Prashant Shenoy. Cut Me Some Slack: Latency-aware Live Migration for Databases. In *Proceedings of the 15th International Conference on Extending Database Technology*, EDBT '12, pages 432–443, New York, NY, USA, 2012. ACM.

[6] Claude Barthels, Simon Loesing, Gustavo Alonso, and Donald Kossmann. Rack-Scale In-Memory Join Processing Using RDMA. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, pages 1463–1475. ACM, 2015.

[7] Carsten Binnig, Andrew Crotty, Alex Galakatos, Tim Kraska, and Erfan Zamanian. The End of Slow Networks: It's Time for a Redesign. *Proc. VLDB Endow.*, 9(7):528–539, March 2016.

[8] B. Cassell, T. Szepesi, B. Wong, T. Brecht, J. Ma, and X. Liu. Nessie: A Decoupled, Client-Driven Key-Value Store Using RDMA. *IEEE Transactions on Parallel and Distributed Systems*, 28(12):3537–3552, Dec 2017.

[9] Chelsio. iWARP: From Clusters to Cloud RDMA. https://www.chelsio.com/wp-content/uploads/resources/iWARP_Then_and_Now.pdf, 2014. [Online; accessed 8-March-2018].

[10] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield. Live Migration of Virtual Machines. In *Proc. NSDI*, pages 273–286, 2005.

[11] Andy Currid. TCP Offload to the Rescue. *Queue*, 2(3):58–65, May 2004.

[12] Sudipto Das, Shoji Nishimura, Divyakant Agrawal, and Amr El Abbadi. Albatross: Lightweight Elasticity in Shared Storage Databases for the Cloud Using Live Data Migration. *Proc. VLDB Endow.*, 4(8):494–505, May 2011.

[13] Aleksandar Dragojevic, Dushyanth Narayanan, and Miguel Castro. RDMA Reads: To Use or Not to Use? *IEEE Data Eng. Bull.*, 40:3–14, 2017.

[14] Aleksandar Dragojević, Dushyanth Narayanan, Orion Hodson, and Miguel Castro. FaRM: Fast Remote Memory. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*, NSDI'14, pages 401–414, Berkeley, CA, USA, 2014. USENIX Association.

[15] Aleksandar Dragojević, Dushyanth Narayanan, Edmund B. Nightingale, Matthew Renzelmann, Alex Shamis, Anirudh Badam, and Miguel Castro. No Compromises: Distributed Transactions with Consistency, Availability, and Performance. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, pages 54–70, New York, NY, USA, 2015. ACM.

[16] Aaron J. Elmore, Vaibhav Arora, Rebecca Taft, Andrew Pavlo, Divyakant Agrawal, and Amr El Abbadi. Squall: Fine-Grained Live Reconfiguration for Partitioned Main Memory Databases. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, pages 299–313, New York, NY, USA, 2015. ACM.

[17] Aaron J. Elmore, Sudipto Das, Divyakant Agrawal, and Amr El Abbadi. Towards an elastic and autonomic multitenant database. In *Proceedings of 6th International Workshop on Networking Meets Databases (NetDB), Athens, Greece*, 2011.

[18] Aaron J. Elmore, Sudipto Das, Divyakant Agrawal, and Amr El Abbadi. Zephyr: Live Migration in Shared Nothing Databases for Elastic Cloud Platforms. In *Proceedings of*

the 2011 ACM SIGMOD International Conference on Management of Data, SIGMOD '11, pages 301–312, New York, NY, USA, 2011. ACM.

[19] Brad Fitzpatrick. Distributed caching with memcached. *Linux journal*, 2004(124), 2004.

[20] P. W. Frey and G. Alonso. Minimizing the Hidden Cost of RDMA. In *2009 29th IEEE International Conference on Distributed Computing Systems*, pages 553–560, June 2009.

[21] Philip W. Frey, Romulo Goncalves, Martin Kersten, and Jens Teubner. A Spinning Join That Does Not Get Dizzy. In *Proceedings of the 2010 IEEE 30th International Conference on Distributed Computing Systems*, ICDCS '10, pages 283–292, Washington, DC, USA, 2010. IEEE Computer Society.

[22] Philip Werner Frey. Zero - Copy Network Communication : An Applicability Study of iWARP beyond Micro Benchmarks. 2010.

[23] Google. Protocol Buffers. https://developers.google.com/protocol-buffers/, 2017. [Online; accessed 3-April-2018].

[24] Juncheng Gu, Youngmoon Lee, Yiwen Zhang, Mosharaf Chowdhury, and Kang G. Shin. Efficient Memory Disaggregation with Infiniswap. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 649–667, Boston, MA, 2017. USENIX Association.

[25] W. Huang, Q. Gao, J. Liu, and D. K. Panda. High performance virtual machine migration with RDMA over modern interconnects. In *Proc. IEEE Int'l Conf. on Cluster Computing*, pages 11–20, 2007.

[26] Anuj Kalia, Michael Kaminsky, and David G. Andersen. Using RDMA Efficiently for Key-value Services. *SIGCOMM Comput. Commun. Rev.*, 44(4):295–306, August 2014.

[27] Anuj Kalia, Michael Kaminsky, and David G. Andersen. FaSST: Fast, Scalable and Simple Distributed Transactions with Two-sided (RDMA) Datagram RPCs. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'16, pages 185–201, Berkeley, CA, USA, 2016. USENIX Association.

[28] David Karger, Eric Lehman, Tom Leighton, Rina Panigrahy, Matthew Levine, and Daniel Lewin. Consistent Hashing and Random Trees: Distributed Caching Protocols

for Relieving Hot Spots on the World Wide Web. In *Proceedings of the Twenty-ninth Annual ACM Symposium on Theory of Computing*, STOC '97, pages 654–663, New York, NY, USA, 1997. ACM.

[29] David Karger, Alex Sherman, Andy Berkheimer, Bill Bogstad, Rizwan Dhanidina, Ken Iwamoto, Brian Kim, Luke Matkins, and Yoav Yerushalmi. Web caching with consistent hashing. *Computer Networks*, 31(11):1203 – 1213, 1999.

[30] Alexey Khrabrov and Eyal De Lara. Accelerating Complex Data Transfer for Cluster Computing. In *Proceedings of the 8th USENIX Conference on Hot Topics in Cloud Computing*, HotCloud'16, pages 40–45, Berkeley, CA, USA, 2016. USENIX Association.

[31] Chinmay Kulkarni, Aniraj Kesavan, Tian Zhang, Robert Ricci, and Ryan Stutsman. Rocksteady: Fast Migration for Low-latency In-memory Storage. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, pages 390–405, New York, NY, USA, 2017. ACM.

[32] H. A. Lagar-Cavilla, J. A. Whitney, R. Bryant, P. Patchin, M. Brudno, E. de Lara, S. M. Rumble, M. Satyanarayanan, and A. Scannell. SnowFlock: Virtual Machine Cloning as a First-Class Cloud Primitive. *ACM Trans. Comput. Syst.*, 29(1):2:1–2:45, 2011.

[33] Feng Li, Sudipto Das, Manoj Syamala, and Vivek R. Narasayya. Accelerating Relational Databases by Leveraging Remote Memory and RDMA. In *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD '16, pages 355–370, New York, NY, USA, 2016. ACM.

[34] Simon Loesing, Markus Pilman, Thomas Etter, and Donald Kossmann. On the Design and Scalability of Distributed Shared-Data Databases. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, pages 663–676, New York, NY, USA, 2015. ACM.

[35] Mellanox. RDMA Aware Networks Programming User Manual. https://www.mellanox.com/related-docs/prod_software/RDMA_Aware_Programming_user_manual.pdf, 2015. [Online; accessed 8-March-2018].

[36] James Mickens, Edmund B. Nightingale, Jeremy Elson, Krishna Nareddy, Darren Gehring, Bin Fan, Asim Kadav, Vijay Chidambaram, and Osama Khan. Blizzard: Fast, Cloud-scale Block Storage for Cloud-oblivious Applications. In *Proceedings*

of the 11th USENIX Conference on Networked Systems Design and Implementation, NSDI'14, pages 257–273, Berkeley, CA, USA, 2014. USENIX Association.

[37] Microsoft. RDMA-capable instances. https://docs.microsoft.com/en-us/azure/virtual-machines/linux/sizes-hpc#rdma-capable-instances, 2017. [Online; accessed 31-October-2017].

[38] Umar Farooq Minhas, Rui Liu, Ashraf Aboulnaga, Kenneth Salem, Jonathan Ng, and Sean Robertson. Elastic Scale-Out for Partition-Based Database Systems. In *Proceedings of the 2012 IEEE 28th International Conference on Data Engineering Workshops*, ICDEW '12, pages 281–288, Washington, DC, USA, 2012. IEEE Computer Society.

[39] M. J. Mior and E. de Lara. FlurryDB: a dynamically scalable relational database with virtual machine cloning. In *Proc. SYSTOR*, 2011.

[40] Christopher Mitchell, Yifeng Geng, and Jinyang Li. Using One-sided RDMA Reads to Build a Fast, CPU-efficient Key-value Store. In *Proceedings of the 2013 USENIX Conference on Annual Technical Conference*, USENIX ATC'13, pages 103–114, Berkeley, CA, USA, 2013. USENIX Association.

[41] Athicha Muthitacharoen, Robert Morris, Thomer M. Gil, and Benjie Chen. Ivy: A Read/Write Peer-to-peer File System. *SIGOPS Oper. Syst. Rev.*, 36(SI):31–44, December 2002.

[42] R. Noronha and D. K. Panda. Designing high performance DSM systems using InfiniBand features. In *IEEE International Symposium on Cluster Computing and the Grid, 2004. CCGrid 2004.*, pages 467–474, April 2004.

[43] Ranjit Noronha and Dhabaleswar K. Panda. Reducing Diff Overhead in Software DSM Systems using RDMA Operations in InfiniBand. 2004.

[44] ORACLE. Delivering Application Performance with Oracle's Infiniband Technology. http://www.oracle.com/technetwork/server-storage/networking/documentation/o12-020-1653901.pdf, 2012. [Online; accessed 31-October-2017].

[45] John Ousterhout, Arjun Gopalan, Ashish Gupta, Ankita Kejriwal, Collin Lee, Behnam Montazeri, Diego Ongaro, Seo Jin Park, Henry Qin, Mendel Rosenblum, Stephen Rumble, Ryan Stutsman, and Stephen Yang. The RAMCloud Storage System. *ACM Trans. Comput. Syst.*, 33(3):7:1–7:55, August 2015.

[46] Marius Poke and Torsten Hoefler. DARE: High-Performance State Machine Replication on RDMA Networks. In *Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing*, HPDC '15, pages 107–118, New York, NY, USA, 2015. ACM.

[47] Jelica Protic, Milo Tomasevic, and Veljko Milutinovic. Distributed Shared Memory: Concepts and Systems. *IEEE Parallel Distrib. Technol.*, 4(2):63–79, June 1996.

[48] An IBM Redbooks publication. Delivering Continuity and Extreme Capacity with the IBM DB2 pureScale Feature. `http://www.redbooks.ibm.com/redbooks/pdfs/sg248018.pdf`, 2012. [Online; accessed 32-October-2017].

[49] W. Rdiger, S. Idicula, A. Kemper, and T. Neumann. Flow-Join: Adaptive skew handling for distributed joins over high-speed networks. In *2016 IEEE 32nd International Conference on Data Engineering (ICDE)*, pages 1194–1205, May 2016.

[50] Oliver Schiller, Nazario Cipriani, and Bernhard Mitschang. ProRea: Live Database Migration for Multi-tenant RDBMS with Snapshot Isolation. In *Proceedings of the 16th International Conference on Extending Database Technology*, EDBT '13, pages 53–64, New York, NY, USA, 2013. ACM.

[51] I. Stoica, R. Morris, D. Liben-Nowell, D. R. Karger, M. F. Kaashoek, F. Dabek, and H. Balakrishnan. Chord: a scalable peer-to-peer lookup protocol for Internet applications. *IEEE/ACM Transactions on Networking*, 11(1):17–32, Feb 2003.

[52] Michael Stonebraker and Ugur Cetintemel. One Size Fits All: An Idea Whose Time Has Come and Gone. In *Proceedings of the 21st International Conference on Data Engineering*, ICDE '05, pages 2–11, Washington, DC, USA, 2005. IEEE Computer Society.

[53] Maomeng Su, Mingxing Zhang, Kang Chen, Zhenyu Guo, and Yongwei Wu. RFP: When RPC is Faster Than Server-Bypass with RDMA. In *Proceedings of the Twelfth European Conference on Computer Systems*, EuroSys '17, pages 1–15, New York, NY, USA, 2017. ACM.

[54] Cheng Wang, Jianyu Jiang, Xusheng Chen, Ning Yi, and Heming Cui. APUS : Fast and Scalable PAXOS on RDMA. Technical report, Department of Computer Science, University of Hong Kong, March 2017.

[55] Kazuo Watabe, Shiro Sakata, Kazutoshi Maeno, Hideyuki Fukuoka, and Toyoko Ohmori. Distributed Multiparty Desktop Conferencing System: MERMAID. In

Proceedings of the 1990 ACM Conference on Computer-supported Cooperative Work, CSCW '90, pages 27–38, New York, NY, USA, 1990. ACM.

[56] Xingda Wei, Sijie Shen, Rong Chen, and Haibo Chen. Replication-driven Live Reconfiguration for Fast Distributed Transaction Processing. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 335–347, Santa Clara, CA, 2017. USENIX Association.

[57] Xingda Wei, Jiaxin Shi, Yanzhe Chen, Rong Chen, and Haibo Chen. Fast In-memory Transaction Processing Using RDMA and HTM. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, pages 87–104, New York, NY, USA, 2015. ACM.

[58] Alex Wiggins and Jimmy Langston. Enhancing the scalability of memcached. *Intel document, unpublished*, 2012.

[59] Erfan Zamanian, Carsten Binnig, Tim Harris, and Tim Kraska. The End of a Myth: Distributed Transactions Can Scale. *Proc. VLDB Endow.*, 10(6):685–696, February 2017.

[60] Wei Zhang, Jinho Hwang, Timothy Wood, K. K. Ramakrishnan, and H. Howie Huang. Load Balancing of Heterogeneous Workloads in Memcached Clusters. In *Feedback Computing*, 2014.