FreeFlow: Software-based Virtual RDMA Networking for Containerized Clouds

Daehyeok Kim

Tianlong Yu¹, Hongqiang Liu³, Yibo Zhu⁴, Jitu Padhye², Shachar Raindel²
Chuanxiong Guo⁴, Vyas Sekar¹, Srinivasan Seshan¹

Carnegie Mellon University¹, Microsoft², Alibaba group³, Bytedance⁴
Two Trends in Cloud Applications

**Containerization**
- Lightweight isolation
- Portability

**RDMA networking**
- Higher networking performance
Benefits of Containerization

Host 1

Container 1

IP: 10.0.0.1

Network App

Software Switch

IP: 30.0.0.1

NIC

Container 2

IP: 20.0.0.1

Network App

Migration

Host 2

Container 2

IP: 20.0.0.1

Network App

Software Switch

IP: 40.0.0.1

NIC

Namespace Isolation ✓

Portability ✓
Containerization and RDMA are in Conflict!

Host 1

Container 1
IP: 10.0.0.1
RDMA App
RDMA NIC
IP: 10.0.0.1

Container 2
IP: 10.0.0.1
RDMA App
IP: 10.0.0.1

Migration

Host 2

Container 2
IP: 20.0.0.1
RDMA App
RDMA NIC
IP: 20.0.0.1

Namespace Isolation ✗
Portability ✗
Existing H/W based Virtualization Isn’t Working

Using Single Root I/O Virtualization (SR-IOV)

Host 1

Container 1

IP: 10.0.0.1

RDMA App

VF 1

IP: 10.0.0.1

RDMA NIC

NIC Switch

Container 2

IP: 10.0.0.2

RDMA App

VF 2

IP: 10.0.0.2

RDMA NIC

NIC Switch

Migration

Host 2

Container 2

IP: 20.0.0.1

RDMA App

VF

IP: 20.0.0.1

RDMA NIC

NIC Switch

VF Virtual Function

Namespace Isolation ✔

Portability ❌
Sub-optimal Performance of Containerized Apps

**RDMA networking can improve the training speed of NN model by ~ 10x !**

- **Speech recognition RNN training**
- **Image classification CNN training**
Our Work: FreeFlow

• Enable high speed RDMA networking capabilities for containerized applications

• Compatible with existing RDMA applications

• Close to native RDMA performance
  • Evaluation with real-world data-intensive applications
Outline

• Motivation

• FreeFlow Design

• Implementation and Evaluation
FreeFlow Design Overview

Native RDMA

Host

RDMA App

Verbs API

Verbs library

NIC command

RDMA NIC

FreeFlow

Host

Container 1

IP: 10.0.0.1

RDMA App

Verbs API

FreeFlow

Verbs library

RDMA NIC

Container 2

IP: 20.0.0.1

RDMA App

Verbs API

IP: 30.0.0.1
Background on RDMA

“Host 1 wants to write contents in MEM-1 to MEM-2 on Host 2”

1. Control path
- Setup RDMA Context
- Post work requests (e.g., write)

2. Data path
- NIC processes work requests
- NIC directly accesses memory
FreeFlow in the Scene

“Container 1 wants to write contents in MEM-1 to MEM-2 on Container 2”

C1: How to forward verbs calls?

C2: How to synchronize memory?
Challenge 1: Verbs forwarding in Control Path

```
struct ibv_qp {
    struct ibv_context *context;
    ...
};
```

```
ibv_post_send (struct ibv_qp* qp, ...)
```

**Attempt 1:** Forward “as it is”
- Incorrect

**Attempt 2:** “Serialize” and forward
- Inefficient
Internal Structure of Verbs Library

```
struct ibv_qp {
    struct ibv_context *context;
    ...
};
```

```
ibv_post_send (struct ibv_qp* qp, ...)
```

Parameters are serialized by Verbs library!
FreeFlow Control Path Channel

**Idea:** Leveraging the serialized output of verbs library

Parameters are forwarded correctly without manual serialization!
Challenge 2: Synchronizing Memory for Data Path

- Shadow memory in FreeFlow router
  - A copy of application’s memory region
  - Directly accessed by NICs

- S-MEM and MEM must be synchronized.

- How to synchronize S-MEM and MEM?
Strawman Approach for Synchronization

“Container 1 wants to write contents in MEM-1 to MEM-2 on Container 2”

Explicit synchronization
High freq.  High overhead
Low freq.  Wrong data for app
Containers can Share Memory Regions

- FreeFlow router is running in a container

MEM and S-MEM can be located on the same physical memory region
Zero-copy Synchronization in Data Path

Synchronization without explicit memory copy:

**Method 1:** Allocate shared buffers with FreeFlow APIs

**Method 2:** Re-map app’s memory space to shadow memory space

FreeFlow supports both!

How to allocated MEM-1 to shadow memory space?
FreeFlow Design Summary

FreeFlow provides near native RDMA performance for containers!
Outline

• Motivation

• FreeFlow Design

• Implementation and Evaluation
Implementation and Experimental Setup

• FreeFlow Library
  • Add 4000 lines in C to libibverbs and libmlx4.

• FreeFlow Router
  • 2000 lines in C++

• Testbed setup
  • Two Intel Xeon E5-2620 8-core CPUs, 64 GB RAM
  • 56 Gbps Mellanox ConnectX-3 NICs
  • Docker containers
Does FreeFlow Support Low Latency?

![Latency Chart]

- **Native RDMA**
- **FreeFlow**

<table>
<thead>
<tr>
<th>Message size (B)</th>
<th>Native RDMA</th>
<th>FreeFlow</th>
</tr>
</thead>
<tbody>
<tr>
<td>64</td>
<td>1.0</td>
<td>0.38μs</td>
</tr>
<tr>
<td>256</td>
<td>1.5</td>
<td>1.46μs</td>
</tr>
<tr>
<td>1K</td>
<td>2.0</td>
<td>2.38μs</td>
</tr>
<tr>
<td>4K</td>
<td>3.0</td>
<td>3.38μs</td>
</tr>
</tbody>
</table>
Does FreeFlow Support High Throughput?

![Graph showing throughput vs message size for FreeFlow and Native RDMA. The graph indicates that FreeFlow's throughput is bounded by the control path channel performance.](image)
Do Applications Benefit from FreeFlow?

![Graph showing CDF of time per step for different methods: Container+TCP, Native RDMA, and FreeFlow. The graph indicates that FreeFlow reduces the time per step by a factor of 8.7 compared to Container+TCP.](image)
Summary

• Containerization today can’t benefit from speed of RDMA.
• Existing solutions for NIC virtualization don’t work (e.g., SR-IOV).

• FreeFlow enables containerized apps to use RDMA.
• Challenges and Key Ideas
  • Control path: Leveraging Verbs library structure for efficient Verbs forwarding
  • Data path: Zero-copy memory synchronization
• Performance close to native RDMA

github.com/microsoft/freeflow