NETWORK VORTEX: DISTRIBUTED VIRTUAL MEMORY FOR STREAMING APPLICATIONS

An Undergraduate Research Scholars Thesis

by

ETA GLUCK

Submitted to the LAUNCH: Undergraduate Research office at
Texas A&M University
in partial fulfillment of requirements for the designation as an

UNDERGRADUATE RESEARCH SCHOLAR

Approved by
Faculty Research Advisor: Dr. Dmitri Loguinov

May 2022

Major: Computer Science

RESEARCH COMPLIANCE CERTIFICATION

Research activities involving the use of human subjects, vertebrate animals, and/or biohazards must be reviewed and approved by the appropriate Texas A&M University regulatory research committee (i.e., IRB, IACUC, IBC) before the activity can commence. This requirement applies to activities conducted at Texas A&M and to activities conducted at non-Texas A&M facilities or institutions. In both cases, students are responsible for working with the relevant Texas A&M research compliance program to ensure and document that all Texas A&M compliance obligations are met before the study begins.

I, Eta Gluck, certify that all research compliance requirements related to this Undergraduate Research Scholars thesis have been addressed with my Research Faculty Advisor prior to the collection of any data used in this final thesis submission.

This project did not require approval from the Texas A&M University Research Compliance & Biosafety office.

TABLE OF CONTENTS

ABSTRACT

Network Vortex: Distributed Virtual Memory for Streaming Applications

Eta Gluck
Department of Computer Science & Engineering
Texas A&M University

Research Faculty Advisor: Dr. Dmitri Loguinov
Department of Computer Science & Engineering
Texas A&M University

Explosive growth of the Internet, cluster computing, and storage technology has led to generation of enormous volumes of information and the need for scalable data computing. One of the central frameworks for such analysis is MapReduce, which is a programming platform for processing streaming data in external/distributed memory. Despite a significant public effort, open-source implementations of MapReduce (e.g., Hadoop, Spark) are complicated, bulky, and inefficient. To overcome this problem, we explore employing and expanding upon a recent a C/C++ programming abstraction called Vortex that offers a simple interface to the user, zero-copy operation, low RAM consumption, and high data throughput.

In particular, this research examines algorithms and techniques for enabling Vortex operation over the network, including both TCP/IP sockets and data-link RDMA (e.g., InfiniBand) interfaces. We developed a new producer-consumer memory stream abstraction presented as a Vortex stream split across two hosts, travelling through a hidden network communication layer to provide the illusion of writing a continuous stream of data directly into a window of memory on a remote machine, thereby enabling the creation of high-performance

networking code and size-agnostic data transport under appropriate circumstances written as simply as an in-memory copy operation, overcoming complications normally inherent in the discrete nature of network packet transfer. While the resulting product is highly workable over standard IP-based internet networks, the design limitations of RDMA technology in interfacing with virtual memory prove to make Vortex streams a suboptimal abstraction for this programming platform, as its central appeal of zero-copy network transfers are rendered largely inaccessible. Alternative algorithms to enhance RDMA performance with Vortex are proposed for future study.

DEDICATION

To my dear friend Mikal, who left us too soon.

ACKNOWLEDGEMENTS

Contributors

My gratitude goes to my faculty advisor, Dmitri Loguinov, for extending to me the opportunity and trust to work with him on this brilliant foray into powerful and safe computing using the Vortex system.

A heartfelt thank you as well to my partners, family, and friends without whom I would be lost, for their love and encouragement both throughout the course of this research and always.

The original source code for "Vortex" used for Network Vortex was provided by Carson Hanel, Arif Arman, Di Xiao, John Keech, and Dmitri Loguinov. A helpful sample codebase demonstrating RDMA networking code was provided to me by Dmitri Loguinov. Servers for testing the derivative code explained herein were provided by the Internet Research Lab at Texas A&M.

All other work conducted for the thesis was completed by the student independently.

Funding Sources

No funding was received for this research.

NOMENCLATURE

Host	A single computer in a network
HCA	Host Channel Adapter
HPC	High-Performance Computing
LAN	Local Area Network
TCP	Transport Control Protocol
IP	Internet Protocol
RDMA	Remote Direct Memory Access

1. INTRODUCTION

The modern computing environment is heavily heterogeneous. From multicore CPUs to distributed systems to even the internet, many of the tasks developers set out to solve today require more than a local program whose effects are isolated to a single host. Web development is its own thriving discipline catering to massively broadcast services, and high-throughput data processing is invariably actualized in the form of distributed systems.

On a low level, however, communication between different hosts is a complicated topic. Layers of protocols and the coordination of data transfer calls for specialized algorithms, most often implemented as literally as possible in low-level languages, with high-overhead abstractions built to accommodate human usage in the high-level languages developers ultimately use.

Network Vortex is a new framework for simple, high-performance streaming networking at a low-level. The library is written in C++ with ease of use in center focus. With it, application developers will be able to write efficient and effective networking code without compromising on performance, allowing for increased complexity and innovation in low- and high-level applications alike.

1.1 Distributed Systems

One application of high-performance networking given special focus in the development of Network Vortex is distributed systems programming, and more specifically, cluster computing.

A distributed system is defined as “a collection of autonomous computing elements that appears to its users as a single coherent system” [1]. In distributed systems programming, data

must be passed fluidly between several devices in order to achieve a unified goal. In high-performance computing scenarios, cluster computing, defined here as a group of similar hosts in a single location connected over LAN set up to parallelize heavy computation beyond what is achievable on a single device, is a type of distributed system with exceptionally demanding requirements on the hosts' ability to synchronize and communicate with one another.

1.1.1 Motivation

Scalability is the primary reason for pursuing a distributed system or cluster computing setup. While the performance of a single server can be improved by upgrading its components—CPU, RAM, potentially GPU(s)—the cost to do so may climb disproportionately higher, with diminishing returns in increased computing power. Additionally, this creates a single, expensive point of failure in the event of a fatal hardware fault.

1.1.1.1 Limits of Hardware

As in the past decades single-core CPUs have plateaued in performance, the need to parallelize computation has become apparent. Multicore CPUs now exist to this end, boasting up to 64 cores (at an incredible price point), after which point compatibility with existing operating systems becomes problematic. GPUs, as a high-end accessory, can handle incredible parallel workloads for some of the simplest of tasks, but their use in more complex algorithmic tasks continues to be limited in practice. Progress in improving the computing power of these devices is steadily ongoing.

Other than such high-end hardware being difficult to acquire, one could foresee limits to these devices as well being discovered in time. In fact, a glaring practical limit is simply obtaining best hardware available. Modern computational workloads already far exceed these specifications; Google's search engine will not run on a single machine no matter if you plug a

64-core, 128-core, nor 256-core processor into it. Unbounded parallelism is a necessity in order to continue to scale.

Distributed systems offer unbounded parallelism. If you get two identical computers, you have twice the computing power. If you get one-thousand identical computers, you have one-thousand times the computing power. No less, each one of these components may be relatively cheap and replaceable, with the entire array capable of being expanded in-place on demand simply by adding yet more computers.

The bottleneck of performance then turns to how well you can synchronize the hosts. Communicating your input and output data to the devices ready to handle them needs to be done as fast and effectively as possible, and this is where Network Vortex plays a role.

1.1.2 Distributed Memory

The main difference for the programmer between utilizing one monolithic system with dozens of cores versus a distributed setup is in accessing memory. While any number of cores on a single host may access the host's central memory to share information, a step of deliberate intervention is necessary to communicate data from the memory of one host to another before setting off a remote routine.

1.1.2.1 The Role of C++

Network Vortex is a C++ library. While this allows it to be used directly in the most performance-critical roles, as well as the development of high-performance high-level frameworks, it also has a more intrinsic purpose to be chosen over a higher-level or functional language. C++ is often considered "low-level" among high-level languages due to it offering a direct, unabstracted[1] view of central memory on a host. Network Vortex, then, assists idiomatic

[1] Up to the domain of the operating system and virtual address space.

C++ programming by extending the programmer's purview with a window to the central memory of a secondary host.

1.2 Vortex

A tunnel in memory between two hosts is achieved using the Vortex framework developed by Carson Hanel, Arif Arman, Di Xiao, John Keech, and Dmitri Loguinov [2]. Vortex is a data streaming solution that operates using a hook into system memory access violations to detect where in an address range a program is currently reading from or writing to, and subsequently triggering some routine to populate empty memory ranges for a reader or to send off freshly written blocks data for a writer, all without any form of manual signaling added in on behalf of the programmer.

1.2.1 Original Vortex

The original derivation of Vortex comes in two notable variants. One is a high-performance sorting routine for use in large datasets. The other, Vortex-C, creates a pair of producer and consumer memory regions for use in parallel algorithms. This one could, for instance, be used to have one thread on a machine generate a large amount of data while another thread writes said data to disk—transparently synchronized with one another such that the data never have to collect completely in memory (and thus capable even of exceeding the size of RAM without breaking down).

The latter of these two is utilized for Network Vortex. Transmitting data from one host to another for processing is, at a high level, a producer-consumer scenario; a large stream of data comes from one host as a producer and is processed by another consumer host as it arrives, splitting the work between the two devices.

2. IMPLEMENTATION

Negotiating data exchange between two hosts without either noticing the exchange occurring inherently calls for multithreaded execution: one independently acting side of a producer-consumer pipeline will act on data as normal, while the other invokes network routines to send or receive the relevant parcels.

2.1 Vortex Usage

To simulate a producer-consumer Vortex-C pipeline split across two hosts, a network data exchange routine is inserted as a hidden intermediary layer, taking over the complementary half of the host's role. In this setup, each machine operates an independent Vortex-C producer-consumer pair. The flow of data from producer to consumer across hosts from the perspective of the producer is then:

1. The producer writes into its local memory stream region,
2. Vortex-C transports the produced data to the intermediary network worker layer as a consumer,
3. A network worker acting as the local machine's consumer sends across all data it receives to a predefined consumer host over the network.

Continuing from the perspective of the consumer host,

4. A network worker acting as the local *producer* continuously reads from a socket, depositing data into the stream as they arrive,
5. Vortex-C transports the received data from the intermediary network layer to the host's intended consumer algorithm, and,

6. The consumer reads the live data coming from its local memory stream, thus synchronizing itself with the producer.

This arrangement allows two hosts to share a stream of memory in the same way a Vortex-C producer-consumer pair allows locally. As two threads running Vortex-C can reach throughput rates of up to 23 GB/s, i.e., over 180 Gbps [2], this could theoretically saturate all but the most high-end data links between two network nodes. The remaining bottleneck to ensure maximum throughput lies in the efficiency of the intermediary layer's network interfacing routine.

2.2 TCP

The first, basic implementation of the network component uses TCP. TCP is the well-refined standard for reliable data transfer used in popular network protocols such as HTTP and can robustly negotiate Network Vortex streams over both LAN and the internet.

TCP is often chosen among application-layer protocols for its robust feature set encompassing reliable data transfer through packet loss detection and retransmission, network congestion avoidance, packet buffering to handle reordering, and flow control, synchronizing both parties in a network transaction.

TCP is critical at the scale of the internet to ensure that shared data transmission channels are utilized fairly, in addition to providing stronger guarantees than IP over unreliable, shared channels. This all means that TCP can flawlessly run Network Vortex across a broad variety of situations.

In addition, flow control provides synchronization between hosts' independent Vortex-C streams. In an unconstrained environment consisting of two disjoint Vortex-C streams on separate hosts, without flow control information being communicated between them, a producer

host would bombard a consumer with packets as fast as it could send them out of its ports. If the consumer algorithm were more time-consuming than the producer, this could lead to significant desynchronization over time and an excessively long work queue on the consumer's side, leading to either memory usage increasing without bound, or high packet loss. TCP's flow control will ensure that the producer host waits for acknowledgement of delivered packets before continuing, corresponding to the rate that the consumer algorithm is draining the operating system's internal TCP buffer, ultimately linking the rate synchronization done in Vortex-C across both hosts.

However, for select high-performance situations, many of TCP's features are unnecessary bloat. Consider cluster computing; should an array of computers be connected on a single LAN subnet, logic to handle packet reordering is superfluous. Furthermore, if hosts each share dedicated links, on appropriate equipment, all of packet loss detection, congestion avoidance, handling reordering, and the entire concept of buffering packets provide no benefit, and only serve as a source of overhead. An alternative to TCP is necessary in these situations.

2.3 RDMA

In the case of stable, dedicated links between hosts, a prime candidate for network communication is the RDMA data-link layer. RDMA, *Remote Direct Memory Access*, is a protocol for a network interface controller that allows user-mode programs to bypass the operating system's networking stack and write directly into main memory [3]. It can skip TCP, UDP, and their associated intermediary buffers, and maximize the rate of data flow across the network.

2.3.1 Who Uses It?

RDMA is used by several large names in computing, such as Nvidia and Microsoft. Microsoft has been developing libraries, protocols, and software interfaces with a focus on

RDMA for years, even continuing development of SMB as *SMB Direct,* featuring RDMA support over a variety of fabrics. Microsoft also developed the Network Direct RDMA interface, which was used in this research's RDMA implementation.

2.3.2 Appropriacy for the Task at Hand

While RDMA can provide perfect raw performance metrics for directly linked machines, it also forgoes the benefit of flow control originally provided by TCP. This means that flow control must be done by hand on the application level, as a part of the networking routine.

This is one reason to begin to consider RDMA a slightly less natural fit at adapting Vortex to a network compared to general-purpose protocols—however, flow control algorithms are widely known and well-studied, and can ultimately be implemented without much difficulty.

However, one of the main draws of RDMA network technology is its *zero-copy* nature, meaning that an application designed around RDMA can tunnel data *directly* out of main memory through a dedicated RDMA-capable network card, or *host channel adapter* (HCA) while avoiding CPU involvement. As with sufficiently meticulous design this may avoid superfluous CPU interrupts and winding paths through the operating system's network stack, this is what often brings superior performance metrics to RDMA networks that cannot be mirrored with TCP/IP.

However, the conditions for this style of networking are not trivial. A major challenge to integrating RDMA with a given system lies in its method of learning memory layouts.

Registering an application for use with RDMA involves three basic steps:

1. Identify the other host, as in any network,
2. Identify yourself, and define your scope, and,
3. Exchange relevant details about your memory environment with one another.

To articulate the *scope* of your application, RDMA requires that you define a memory region, associated with a buffer. In order to act on said memory region, RDMA then *pins* all mentioned pages. Pinning a page in memory freezes the mapping between logical addresses, in the language of your program, with a particular *physical* page configuration in the language of the RDMA HCA. This is problematic for three reasons:

1. If your program encompasses a particularly broad area of memory, RDMA will force this to become either a contiguous, monolithic (fully-allocated) buffer, or a collection of disjoint, potentially ad-hoc memory regions, with hindered capability for co-operation between them.
2. If the areas of interest in your program are continuously moving, it becomes exceptionally difficult to target any spot to become a memory region, and it becomes *impossible to reuse them*.
3. If your program requires virtual memory or unusual paging, RDMA cannot work as intended.

These unfortunately affect Vortex threefold. Vortex, as a streaming application, is designed for use with large expanses of memory that are impractical through straightforward approaches. Furthermore, the almost completely linear movement of such scenarios means that you may fundamentally never use the same area of memory twice. And finally, Vortex relies almost completely on virtual memory and paging. Pinning a page in the middle of a Vortex stream simply breaks the framework, as it can no longer reuse or conserve memory as it is designed to. Pinning the pages ahead of or behind a Vortex stream is a gross waste of resources, and ultimately reduces you to the efficiency of malloc.

Accordingly, while many high-performance programs are well suited to for the offerings of RDMA and may reap substantial benefits from it, Vortex happens to be one of the least possible equipped to benefit from it. So, how does this work out in practice?

3. BENCHMARKS

There are three primary dimensions to measure the effectiveness of the implementation, based on its three design goals:

1. Ease of Use

   The subjective knowledge and additional lines of code required to add networking functionality to a program using Network Vortex.

2. TCP Performance

   The peak transfer speeds achieved by the implementation running through the OS networking stack with TCP.

3. RDMA Performance

   The peak transfer speeds achieved by the implementation running through a dedicated InfiniBand link using RDMA.

The original focus of the Network Vortex implementation is the optimization of dimensions **1** and **3**, for suitability in high-performance computing. Dimension 2 is also considered, and has value for enabling operation over the general internet. However, it is not the focus of this research, and can be expected to take readily to improvement in further work.

3.1 Ease of Use

The first and arguably most important factor by which we measure a successful implementation is the ease with which a programmer can incorporate the Network Vortex framework into the architecture of a program to complete a task requiring networking. The framework should:

1. Not require coordination for when to send intermediate segments of data,
    a. Not require a detailed understanding of packets, maximum segment size, or serialization,
2. Operate seamlessly in an (otherwise) single-threaded program, and
3. Ideally, require the understanding of fewer than 10 lines of code to operate a connection.

When these criteria are all met, operating a Network Vortex connection from either side in C++ should be as easy as entering an IP address and then typecasting a pointer to read and write data at a specified location; one step more than standard Vortex. Regardless of experience in networking or time available to the programmer, one shall then have the toolset to make an efficient networked application.

```
#include <cstdint>
#include <cstdio>
#include <numeric>

void read(const std::uint64_t *from, std::size_t size) {
    const auto sum = std::accumulate(from, from + size, 0ull
);
    std::printf("Received sum: %llu\n", sum);
}

void write(std::uint64_t *into, std::size_t size) {
    std::iota(into, into + size, 0ull);
    const auto sum = (size - 1) * size >> 1u;
    std::printf("Delivered sum: %llu\n", sum);
}
```

Figure 3.1: Simplicity of an ideal networked data transfer.

To benchmark the simplicity of dataflow code, Figure 3.1 displays two basic functions for populating and summing an array, respectively, in idiomatic C++ code, using the C++ standard library and iterator pattern. If these two functions can be used unchanged with Network Vortex streams with 10 or fewer lines of connection overhead, this goal is considered to be

satisfied, as they are already intentionally written as simple as possible, with zero knowledge of networking protocols in mind, and as such is one of our benchmarks.

3.2 TCP Performance

TCP is the only protocol explicitly supported by the framework capable of operating over the internet at large. As such, it is important that its overhead in this regard be competitively low versus other high-level networking libraries. For this purpose, the TCP implementation can be seen as providing an *easy*, though not necessarily HPC-grade, networking framework in C++.

90% to 95% of the maximum feasible throughput of typical TCP networking software would then make this an excellent tool for general networking use.

3.3 RDMA Performance

As a datacenter-grade, high-performance networking technology and the focus of this research implementation, the RDMA implementation of Network Vortex should be as fast as hardware allows. The goal for RDMA performance is low enough overhead for data transfer throughput to be capable of saturating an InfiniBand connection (above 95% of available bandwidth).

4. RESULTS

The implementation of Network Vortex provided in the accompanying library effectively achieves the two of the three goals set out in section 3.

4.1 Ease of Use

The sample functions shown in Figure 3.1 are operable with the Network Vortex library in the following minimal reader and writer configurations (see Appendix A for the specification of `NetworkWrapper`):

```
void reader() {
    const char *senderAddress = "127.0.0.1";
    NetworkWrapper net;
    char *stream = net.openReceiver(senderAddress, "40104",
                                    nullptr, 22, 1, 1, 2);
    if (stream == nullptr) {
        std::puts("Could not open reader");
        std::exit(-1);
    }
    read((std::uint64_t *) stream,
         net.getSize() / sizeof(std::uint64_t));
    net.waitUntilDone();
}

void writer() {
    NetworkWrapper net;
    const std::size_t transmissionLength = 1ull << 35u;
    char *stream = net.openSender(40104, transmissionLength,
                                  nullptr, 22, 1, 1, 2);
    if (stream == nullptr) {
        std::puts("Could not open writer");
        std::exit(-1);
    }
    write((std::uint64_t *) stream,
          transmissionLength / sizeof(std::uint64_t));
    net.closeSender();
}
```

Figure 4.1: Runner code for simple data transfer.

The routine as displayed sends 32 GiB of data through the loopback address 127.0.0.1 on port 40104 in fewer than 10 lines per side, with a significant block thereof taken for

straightforward error checking— which itself does not constitute a conceptual barrier for adopting the framework. It is reasonable to assert that after setting up appropriate header files, with the example in Figure 4.1. as a “quick-start,” one may get a first simple application running Network Vortex configured in as little as 30 minutes

However, the functions provided in Figure 3.1 represent only the most basic of data transfer. Indeed, a non-trivial application to send files over a network may also be implemented with ease using the framework:

```
void read(const char *from, std::size_t size, const char *
outputFilename="received.dat") {
    std::printf("Expecting %llu bytes\n", size);
    auto file = std::fopen(outputFilename, "wb");
    auto totalTransferred = 0ull;
    const auto b = 2048;
    while (totalTransferred < size) {
        const auto nextTransferSize = min(b, size -
totalTransferred);
        totalTransferred += std::fwrite(from +
totalTransferred, 1, nextTransferSize, file);
    }
    std::fclose(file);
}

void write(char *into, std::size_t size, const char *
inputFilename) {
    auto file = std::fopen(inputFilename, "rb");
    std::vector<char> localSource(size);
    std::printf("Transferred %llu bytes\n", fread(localSource
.data(), 1, size, file));
    std::fclose(file);
    std::memcpy(into, localSource.data(), size);
    std::puts("Written");
}
```

Figure 4.2: (Imperfect) reading and writing functions for basic file transfer.

The runner code to operate a full program from the data transfer functions in Figure 4.2 can be found in Appendix B. The necessary changes consist of first calculating a transfer size based on file size, and then simply using the `char` stream as normal, without casting to an

integral type. However, for completeness, the runner code provided in Appendix B presents a fully-featured peer-to-peer file transfer program written in under 70 lines of C++ code, including the routines provided in Figure 4.2.

This also demonstrates one weakness regarding ease of use with Network Vortex (common to the use of Vortex in general), in that occasional operating system functions will behave strangely with protected memory regions used by Vortex and require basic buffering. This code may not run at maximal efficiency as a result; however, base Vortex comes with file reader and writer stream types which may be integrated with this code if performance is desired. However, this entire design goal was accomplished due to the beauty of Vortex's original interface, and this further goes to show how effective of a framework Vortex is overall.

4.2 TCP Performance

The TCP implementation of Network Vortex is both robust, capable of completing data transfers over long-distance internet connections, as well as performant. The exact details of the testing servers are listed in Table 4.1.

Table 4.1: Testing Servers.

	Server A	Server B
CPU	i7-4930K	Xeon E5 2680 v2
Architecture	Ivy Bridge-E	Ivy Bridge-EP/EX
Cores	6	10
Clock Speed	3.4 GHz	2.8 GHz
RAM	32 GB	192 GB
RAM Type	2400 MHz DDR3	1866 MHz DDR3
OS	Server 2016 Datacenter	Server 2016 Datacenter
Ethernet Link	Intel 82579LM Gigabit	Intel 10G 4P X540
RDMA Interface	HCA Mellanox ConnectX-3 40 Gbps IPoIB Adapter	*None*

Over the limiting 1 Gbps interface, Vortex over TCP achieved a throughput of 935 Mbps on a simple `memset` benchmark. In comparison, the popular network benchmarking tool iPerf3 achieved 948 Mbps over the same link. This puts Vortex over TCP at 98.6% the efficiency of a well-optimized benchmark and at 93.5% the advertised efficiency of a common consumer network link—which is outstanding provided the goals of this research. However, it is likely that the performance would not scale at higher bandwidths due to the previously discussed limitations of TCP.

4.3 RDMA Performance

As only one of the servers involved was capable of using RDMA, tests were conducted over RDMA loopback. As RDMA loopback (unlike TCP/IP loopback) takes approximately the same path through the network card as a regular transfer, this adds only a slight advantage.

Using a customized RDMA project as a benchmark, peak rates of approximately 43.8 Gbps (above 40 Gbps due to loopback) even with flow control added in were observed before being integrated with Vortex, with the benchmarking software `nd_send_bw` reporting 46.13 Gbps on the same connections.

However, a straightforward migration of the same benchmark and flow control to the Vortex library cut its performance in half, only reaching a maximum of 23.80 Gbps.

4.3.1 Problems

There are two main performance issues encountered when adding RDMA as a network layer between hosts: copying and pipelining. In the straightforward implementation, a single memory region is registered for RDMA transfers on each host, and memory is copied in and out of it so that the network hardware may perform transfers with only static address information. This incurs a memory copy, which itself should outpace a 40-gigabit link, though adding on CPU usage—however, as single-threaded, single-buffer setup, this must constantly alternate between copying and waiting on network transfers, so the link cannot be utilized at all times. An ideal implementation would always have data ready to transfer so that the network hardware can remain at 100% utilization. Two possible solutions are proposed next.

4.3.2 Improvements

There are two algorithms that may vastly increase the throughput of Vortex over RDMA.

4.3.2.1 Zero-Copy

One of the limitations of RDMA hardware is the requirement that registered memory regions stay pinned throughout a transfer. As such, an entire Vortex stream may not be registered, but micro-regions within it may.

The Zero-Copy algorithm creates as many threads as accessible comeback blocks (dictated by the Vortex-C parameters *M* and *L*). Each of these threads is assigned to a block and executes the following routine:

1. Check if the block is ready for transfer.
    a. Similar to TCP, this waits for the block furthest back in the comeback region to finish processing before continuing to avoid Vortex-C moving the page associated with it away before it is ready.
2. Register the block as a memory region
3. Exchange memory region details associated with the block number with the other host through a control thread
    a. The control thread holds a synchronized queue of messages containing details on newly registered memory regions on the local host, and transmits them over to the other host whenever one becomes available.
    b. The control thread also awaits messages from the remote host, and marks each local block as ready-for-transfer when its remote counterpart's info is received, waking a local thread if one is already waiting.
4. Wait for a ready-for-transfer indication, then send or receive the block.
5. Await completion of the transfer, deregister the memory region, then move to the next block in the stream with no worker thread present.

6. Repeat.

Sleeping while awaiting ready-for-transfer indications can be achieved via `WaitOnAddress` in the Win32 API, or C++ atomic variables more generally. The cost of registering and deregistering memory regions is the main obstacle of this algorithm. However, the overhead of memory registrations is less than the overhead of copy operations at block sizes above 256 KB [4].

4.3.2.2 Pipelined-Copy

Since memory copies are quite fast, a second approach is to avoid alternating off of network sends by pipelining blocks to send across several buffers, while keeping the idea of static buffers. The Pipelined-Copy algorithm operates similarly to the Zero-Copy algorithm, first allocating as many buffers as threads.

1. Check if the block is ready for transfer.
    a. Identical to the Zero-Copy algorithm.
2. Let a control thread operate a queue to constantly initiate asynchronous transfers on ready buffers
    a. For the sender, this is a multiple-producer, single-consumer queue, where worker threads wait for the control thread to consume (transfer) existing data in each buffer.
        i. Senders start at a block and wait for an empty buffer index into which to transfer data.
    b. For the receiver, this is a single-producer, multiple-consumer queue, where worker threads wait for the control thread to produce data in each buffer to transfer into the stream.

i. Receivers wait for a full buffer index and block index and jump to the corresponding block's memory address in the stream to move the buffer.

3. Perform the memory copy of the thread's assigned block to an open buffer (sender), or from the thread's assigned buffer to its associated block (receiver).
4. Repeat.

This is a simpler algorithm and should allow for full utilization of a 40 Gbps link, but its primary drawback is requiring copies, thus scaling with CPU performance and network hardware rather than network hardware alone and incurring significant CPU utilization. The time overhead of copying should however be preferable to page registration at block sizes of less than 256 KB, though smaller block sizes may also lead to a decrease in performance overall due to increased frequency of required synchronization [4].

5. CONCLUSION

The operation of Vortex over a network is robust and effective for TCP/IP purposes. For high-performance networking using RDMA, there are better choices available at present.

However, further research to pursue the proposed optimizations to an RDMA network transfer layer may yet produce a datacenter-grade technology. The Zero-Copy algorithm, if possible, may allow for performance that scales perfectly with RDMA hardware, while the Pipelined-Copy algorithm will be robust with few unexpected interactions. It is expected that these may reach link utilizations of 95% or higher, depending on the hardware and software context.

REFERENCES

[1] M. van Steen and A. S. Tanenbaum, “Introduction,” in *Distributed Systems*, 3rd ed., Maarten van Steen, 2017, pp. 2–2.

[2] C. Hanel, A. Arman, D. Xiao, J. Keech, and D. Loguinov, “Vortex: Extreme-Performance Memory Abstractions for Data-Intensive Streaming Applications,” in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, New York, NY, USA: Association for Computing Machinery, 2020, pp. 623–638. [Online]. Available: https://doi.org/10.1145/3373376.3378527

[3] Rajkumar Buyya; Toni Cortes; Hai Jin, "An Introduction to the InfiniBand Architecture," in High Performance Mass Storage and Parallel I/O: Technologies and Applications, IEEE, 2002, pp.616-632, doi: 10.1109/9780470544839.ch42.

[4] P. W. Frey and G. Alonso, “Minimizing the Hidden Cost of RDMA,” in *2009 29th IEEE International Conference on Distributed Computing Systems*, 2009, pp. 553–560. doi: 10.1109/ICDCS.2009.32.

[5] K.-H. Lee, Y.-J. Lee, H. Choi, Y. D. Chung, and B. Moon, “Parallel data processing with MapReduce,” ACM SIGMOD Record, vol. 40, no. 4, pp. 11–20, 2012, doi: 10.1145/2094114.2094118.

[6] V. Kalavri and V. Vlassov, "MapReduce: Limitations, Optimizations and Open Issues," 2013 12th IEEE International Conference on Trust, Security and Privacy in Computing and Communications, 2013, pp. 1031-1038, doi: 10.1109/TrustCom.2013.126.

APPENDIX A: NETWORKWRAPPER INTERFACE

A portion of the library implementation for Network Vortex is provided in this section. As a reproduction of licensed code, a mandatory copyright notice is included below:

```
#pragma once
#ifdef _WIN32

#include <memory>

class NetworkWrapper {
    char *buf{};  // Managed end of `s`
    SOCKET connection{};
    uint64_t streamSize{};
    VortexC *s{};
    std::unique_ptr<VortexC> managedStream{};

    static bool wsaInitialized;

    enum class Mode {
        receiver, sender, unassigned
    } mode = Mode::unassigned;
    thread worker{};

    uint64_t blockSize{};

    void run();
    void transfer(uint64_t offset);

public:
    NetworkWrapper();
    ~NetworkWrapper();

    char *openReceiver(const char *senderAddress, const char
 *senderPort, VortexC *sExternal, int blockSizePower, int L,
int M, int N);
    char *openSender(unsigned short senderPort, uint64_t size
, VortexC *sExternal, int blockSizePower, int L, int M, int N
);

    void closeSender();

    uint64_t getSize() const { return streamSize; }
    VortexC *getStream() { return s; }

    void waitUntilDone();
};

#endif
```

Figure A.1: The header file interface for `NetworkWrapper`, the central class of Network Vortex.

Public methods:

- `openReceiver`: Opens a client receiver connection to pair with a ready sender server. After calling `openSender` on a host with data to send, this method should be called with the IP address of the host and a matching port number to initiate a connection. Returns the readable Vortex stream associated with the network connection.
- `openSender`: Opens a server listening for connections on all network interfaces and advertising the transfer length. After receiving a connection request from a corresponding `openReceiver` call, this returns a writable Vortex stream associated with the network connection.
- `closeSender`: Commits and sends the last segment of data from the sender's side, called manually after all data has been written. Corresponds to a Vortex stream's `FinishedWrite` method.
- `getSize`: Retrieves the transfer length declared by the sender in the connection pair.
- `getStream`: Retrieves another handle to the memory address normally returned by `openSender` or `openReceiver`.
- `waitUntilDone`: Frees stream resources on a reader host, called manually after exhausting a stream.

APPENDIX B: FILE TRANSFER RUNNER CODE

```
std::size_t filesize(const char *filename);

int main(int argc, char **argv) {
    NetworkWrapper net;
    if (argc != 3) {
        printf("usage: %s <r sender-address | s filename>\n"
, argv[0]);
        return 1;
    }
    else if (tolower(argv[1][0]) == 'r') {
        char *stream = net.openReceiver(argv[2], "40104",
                                        nullptr, 22, 1, 1, 2);
        if (stream == nullptr) {
            std::puts("Could not open reader");
            std::exit(-1);
        }
        read(stream, net.getSize());
        net.waitUntilDone();
        std::puts("Closed");
    } else if (tolower(argv[1][0]) == 's') {
        const std::size_t transmissionLength = filesize(argv[
2]);
        char *stream = net.openSender(40104,
transmissionLength,
                                      nullptr, 22, 1, 1, 2);
        if (stream == nullptr) {
            std::puts("Could not open writer");
            std::exit(-1);
        }
        write(stream, transmissionLength, argv[2]);
        net.closeSender();
        std::puts("Finished");
    }
    return 0;
}

std::size_t filesize(const char *filename) {
    auto file = std::fopen(filename, "rb");
    std::fseek(file, 0l, SEEK_END);
    const auto size = std::ftell(file);
    std::fclose(file);
    return size;
}
```

Figure B.2: Full runner code to operate the functions presented in Figure 4.2. Combined, this constitutes a full peer-to-peer file transfer program in under 70 lines of C++ code.