Vortex-topology Messaging: Data Vortex for high performance middleware

Arno Kolster, Providentia Worldwide, San Francisco, CA, USA
S. Ryan Quick, Providentia Worldwide, Fredericksburg, VA, USA

Abstract

Data Vortex has made waves in the HPC space for several years, but the technology has wider applications in other aspects of computing. Large enterprise distributed systems rely on messaging middleware for efficient communications and remote procedure calls. The open source community leverages messaging middleware under some of the largest hyperscale systems including cloud management platforms, big data event processors, distributed analytics systems, and social media and messaging engines. With an affinity for fixed-length messages and an egalitarian topology, Data Vortex lends itself to the role of messaging transport for a variety of use-cases.

This paper explores the application of the Data Vortex network topology for messaging middleware. The study outlines the exploration of the popular RabbitMQ messaging broker and message relay, delivery, and manipulation with the data vortex as transport. The paper examines the effort needed to bring Erlang/OTP and the RabbitMQ messaging middleware to Data Vortex and looks at initial performance comparisons with other network topologies in the same roles. Extending this detail, the work looks at the next stages for testing and likely best-practice applications for the Data Vortex in messaging middleware systems given the initial data from RabbitMQ.

Keywords: data vortex, messaging middleware, rabbitmq, high performance computing

Introduction

As the lines between high performance computing (HPC) and hyperscale architectures blur, we see the need for technology to bridge gaps and uplift areas for which those technologies were not originally intended. As the need for computational solutions to large problems and ever larger datasets grows, so does the need for efficient and optimized communications protocols and behaviors between components. While there has been tremendous growth in highly scalable networking systems in a variety of contexts, most of those efforts have focused on delivering parallel computing applications efficiently and with reduced latency.

One often overlooked space for optimization, however, is in generic messaging between disparate applications and frameworks. This is the paradigm underlying most hyperscale workloads and particularly beneath microservice systems architectures. While there has been an explosion in messaging middleware as a part of the BigData wave – bringing a wide variety of new commercial and open source implementations – there have not been large improvements on optimizing the underlying transport directly for a dedicated messaging workload. Improvements in high performance interconnects optimize for parallel computing cases and not for the generic publisher and consumer architectures needed in loosely coupled distributed computing workloads.

Data Vortex is a new player in the high performance interconnect space, with impressive results for GFFT, HPCC, Graph500 Breadth First Search[1], and other parallel computing benchmarks. We want to explore the viability for Data Vortex in this traditional messaging middleware space for disparate communications between de-coupled publishers and consumers with a variety of common messaging patterns. Data Vortex topology is optimized for small-packet sizes and provides an egalitarian and uniform scaling profile as nodes increase. This suggests a role for optimizing general-purpose messaging systems to make use of the Vortex topology for message delivery. This paper looks at Data Vortex in this light, by providing Data Vortex interfaces to a common open source message broker (RabbitMQ) and looking at the effort involved, initial performance data, and suggested next steps to continue the research.

1 Test Design and Rationale

We looked at a variety of common messaging patterns for streaming analytics, service-oriented and microservices architectures popular in hyperscale environments. One of the most common problems we see is in the scalability of messaging middleware in very large deployments where message properties vary by use-case and consumer. This varies widely from the traditional parallel computing effort with players like MPI because many of the characteristics of the message are not determined at creation but by the consumption of the message, and where publisher and consumer are tightly coupled. For general purpose, high performance and hyperscale messaging share common requirements like low-latency and high transport bandwidth. But the hyperscale use-cases require that consumers come and go without impact (or even notification) to publishers; both ordered and un-ordered delivery for the same message stream; a variety of durability attributes depending on the consumer, location, number of transport hops; disparate security controls, etc. To date, most messaging middleware software focuses on a subset of these requirements and generally relies on the performance of the transport layer only for latency and bandwidth.

1.1 Rationale for Data Vortex

[1] https://www.datavortex.com/performance/

Since the Data Vortex provides a uniform, linear scaling model as nodes and switches increase, this could allow for some additional message requirements to move into the transport layer. For example, ordered message delivery could be enforced by the network itself and not rely on enveloping, multicast command and control announcements, and sliding windows. The goal of the project then, is to demonstrate the efficacy for the Data Vortex (DV) system for handling these message patterns as a possible avenue for development for both the platform and for messaging systems optimization in the future. Since this effort aims to solve problems which exist in current commercial and open source general purpose messaging middleware, we decided that augmenting one of the common players in this space would be a good starting point for the project and allow for immediate comparison with a large base of extent performance and configuration data from large deployments.

1.2 Rationale for RabbitMQ

RabbitMQ, an open source message broker implementing the Advanced Message Queueing Protocol (AMQP) and other protocols, is well-known for its ubiquity, ease of installation and configuration, and scalability in a variety of common deployment architectures. Implemented using the Open Telecom Platform (OTP) in Erlang, the platform is extensible and supports very high performance deployments for distributed computing systems. Designed for true general purpose messaging, RabbitMQ supports configuration for many messaging patterns and disparate durability and delivery characteristics as well.

Because of its ubiquity and extensibility, we chose the platform to test for Data Vortex. The general project design is to deliver messages between RabbitMQ clusters using DV for transport, and to compare the initial results with another high performance interconnect: Full Data Rate (FDR) Infiniband.

1.3 Test Harness Design and Implementation

Figure 1 shows the structure for the testing. Leveraging 4 nodes of an 8 node Data Vortex machine and the management node for the Data Vortex system, we construct two 3-node RabbitMQ clusters. The clusters are configured to provide AMQP messaging exchanges and queues on the DV nodes and configuration, management, and reporting on the DV management node. Docker containers provide standardized RabbitMQ and Erlang deployments across all nodes[2].

[2] Detailed configuration data – including configuration files, container construction scripts, all results files, and code for all of the project tools and tests are available through the Jupyter notebook for the project. See the references section for details on accessing the notebook.

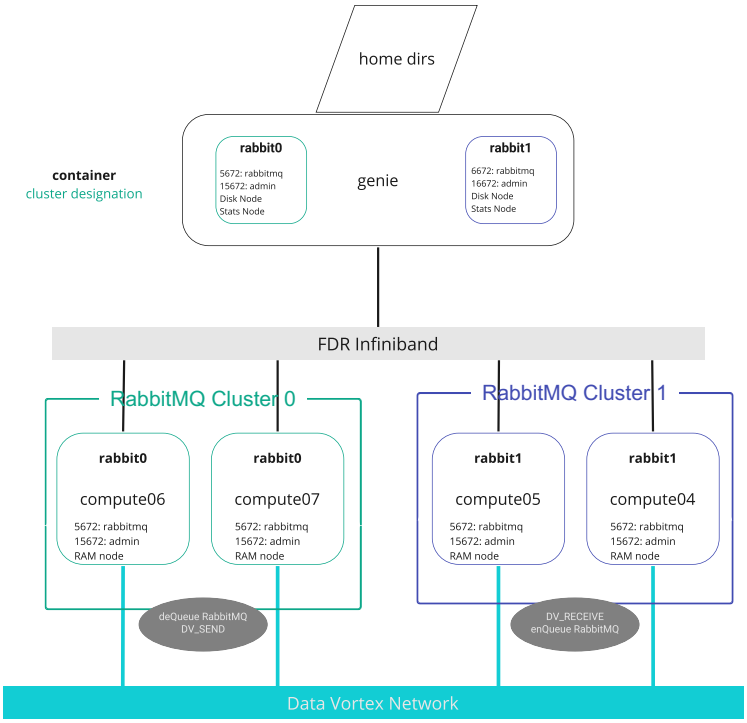


Figure 1: Data Vortex Test Harness

1.3.1 H3 Test Harness Message Flow

Using a range of message sizes (8, 16, 32, 64, 128, 256, 512, 1024, 2048, 4096, 8192, 16384, 32768, 65536, 131072 bytes) measure performance sending messages across the network and storing into RabbitMQ queues. RabbitMQ is configured using topic exchanges with distributed queues according to the layout in Figure 2. Queues are non-persistent and distributed using subtopics, where ".A" queues are on node1 and ".B" queues are on node2. The topic exchange configuration allows AMQP delivery on both of the cluster nodes. Each transport type under testing uses a dedicated exchange with like queue configuration beneath. One additional node (in our testing, this is the systems management node for the Data Vortex system) provides RabbitMQ command and control, configuration, management, and statistics collection and display functionality.

Functional flow for the system under test (SUT) works as follows:

1. Construct messages on sending nodes. For the SUT, these are the brokers in the other cluster, e.g. rabbit1 cluster sends to rabbit0 cluster.
2. Send messages to receiving brokers across the transport under test (Infiniband, DV).
3. Record and persist message delivery performance across the transport and into the RabbitMQ queue.
4. Confirm that the message delivered matches the message sent (bytes, md5sum)
5. Reset AMQP brokers to clear available memory and prepare for the next run.

Specific tests are outlined in the sections below and are repeated for both Infiniband and Data Vortex.

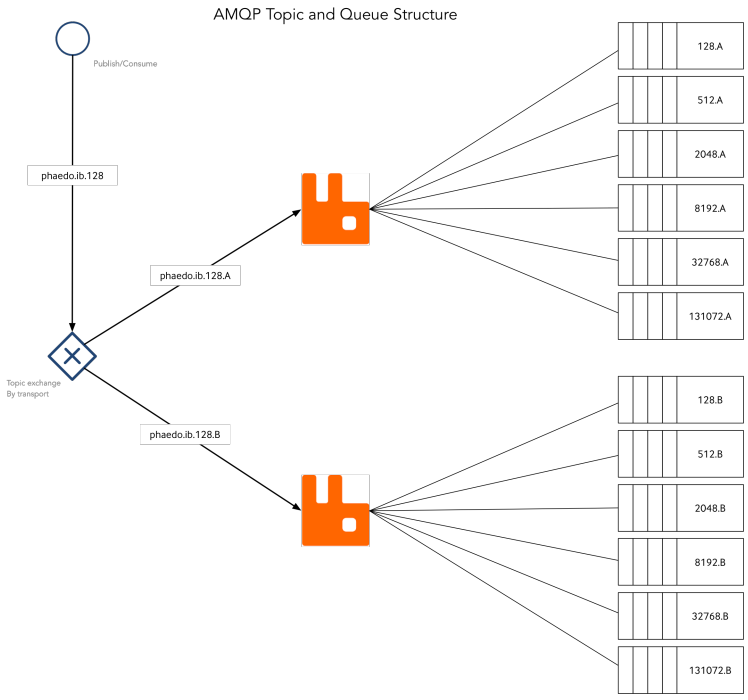


Figure 2: AMQP and Broker Exchange and Queue Structure

1.3.2 H3 Test Harness Implementation Details

Since the nature of the test analyzes network behavior and performance, we deliberately avoided any persistence in the message path. This required large memory footprints available both to our SUT software and to RabbitMQ brokers to hold and account for messages during the test runs. As a result, the number of messages transported during each test decreases as the message size changes. This conserves available memory and ensures that we run the tests long enough to reach steady state, but that we would not exhaust all available memory resources on the nodes. We felt strongly that we needed to run the tests for a longer period of time, and did not want to involve a potential bottleneck from slow queue draining on the consumer side. This meant we terminated the testing with messages still in the queues on the receiving brokers. We then drained and reset queues between each run to clear memory for new messages and to reset garbage collection within the erlang VM.

Table 1 shows the message sizes, number of test iterations, and number of messages sent to topic exchanges and queues in each iteration.

Message Size (bytes)	Number of Messages Delivered[3]	Test Iterations
8	20,000,000	2
16		
32		
64		
128		
256		
512		
1024		
2048	10,000,000	
4096		
8192		
16384	4,000,000	4
32768	3,000,000	
65536	1,700,000	6
131072	600,000	8

Table 1: Test Run Configuration

2 Software Test Implementation and Feature Parity Analysis

The Data Vortex API is written for parallel computing. While expected, this means that there are expectations and assumption with regards to API usage in the client application which make some integration efforts like ours more difficult. The DV API structure assumes a synchronicity between publisher and consumers by node and message, for example. This means that it is not simple for to attach additional publishers and consumers at will because running DV applications define these parameters for sending and receiving at initialization. This is not a barrier to implementing general messaging paradigms, but we call it out here since it had implications on the sorts of message patterns we could implement readily for the study. We elected not to implement true publish/subscribe or affinity subscription messaging for this test and instead utilized queued messages with brokers as message final destinations. While we did implement consumers outside the SUT, their purpose was simply to remove messages from queues and to confirm that the messages matched end-to-end. The performance of these consumers is not recorded or reported for this project.

The DV topology leverages 64-bit words which can be aggregated by chunking for delivering larger messages. Where RabbitMQ does not limit message size (beyond the capabilities of the underlying operating system) and permits variable message sizes in queues and exchanges, the

[3] The number of messages changes here solely as a function of available memory consumption on the receiving broker nodes. Our DV system had 100G of memory allocated for RabbitMQ. Running over memory allocation results in RabbitMQ invoking automated persistence which skews results. For this reason, we deliberately made the decision to manipulate memory by message count and adjusting the RabbitMQ high-watermark for choosing to invoke message persistence in the cluster.

Data Vortex does not. To implement variable size tests we ran independent tests of fixed sizes. Implementing variable length messages for Data Vortex is possible, but the scope for that effort went beyond what was reasonable development for this analysis. The DV linear scaling model makes the performance (once measured) fully predictable however, so this should not affect performance if variable-length messaging comes to the API.

Communications with RabbitMQ (sending and receiving) leverages TCP sockets. We accepted this limitation for our testing as well and deliver messages into RabbitMQ exchanges using the loopback TCP binding after the message has already moved across the network for DV. Since Infiniband implements a full TCP/IP stack (IPoIB), those tests connected from the source brokers to the destination brokers using a standard networked TCP connection. So the detailed message flow between nodes differs slightly between transport implementations as illustrated in Figure 3.

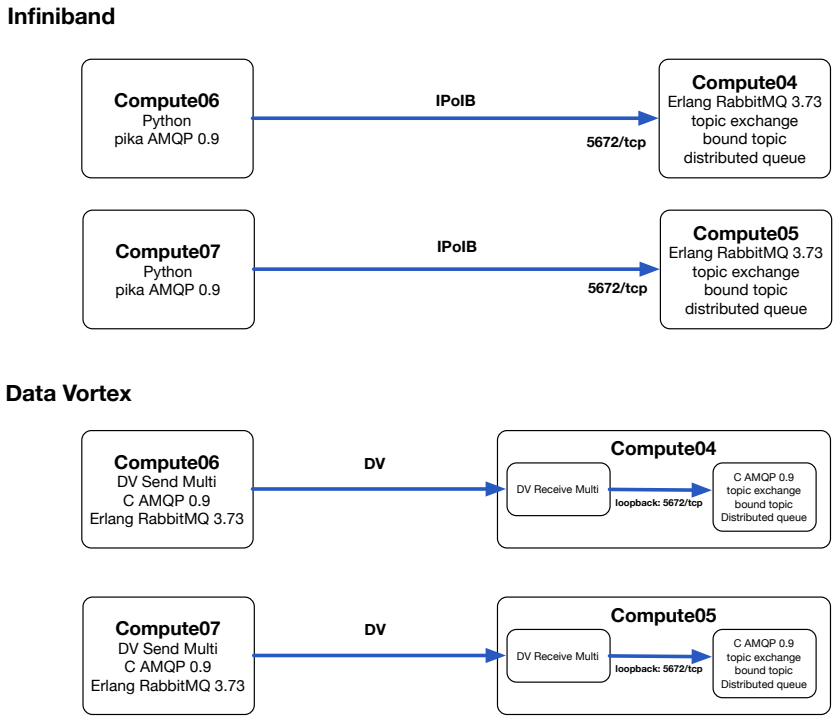


Figure 3: SUT Transport Implementation

2.1 Software Development for Data Vortex

Overall, the DV API is straightforward and implemented according to published design. Available only in C, however, means that modern integrations from higher level and functional languages will require C bindings or additional effort to provide native APIs for those languages. For our test implementation, we chose to leverage a C-native AMQP library for interfacing with RabbitMQ on the same node rather than working to develop directly in Erlang. This allowed the team to implement the DV and AMQP calls directly without the added overhead of additional Erlang-to-C bindings.

2.2 Feature Parity for Open Source Messaging Middleware

2.2.1 Asynchronous Architectures

Currently, loosely-coupled, asynchronous applications using the DV API is not possible. This did not hinder our efforts once we understood the situation, but substantial changes would be required for a production implementation with a scale-out architecture. The hardware design of Vortex Interface Cards (VICs) and switches does not presume this tight coupling however, so the effort to provide for general messaging and independent consumers and producers should be straightforward for a future release of the Data Vortex API.

2.2.2 Software Development Feature Parity

Most modern open source messaging systems leverage java, scala, python, nodeJS, go, or erlang. Most high performance computing (HPC) parallel messaging applications utilize C, C++, or Fortran. Since Data Vortex has its pedigree in HPC systems and applications, it lacks direct support for higher level languages and cloud native design patterns. We deliberately implemented the RabbitMQ brokers in Docker containers and used Python for the Infiniband transport tests as a means to evaluate how well the Data Vortex hardware integrates with hyperscale development techniques and common software deployment mechanisms.

The Data Vortex system under test worked well with interactions between node-local applications and containerized applications (like RabbitMQ). Likewise, performance for bridged containers and the interaction between the C DV API and containerized AMQP socket communications was very good. We did not detect any bugs in either the API or in the container configurations. Our initial configurations for Docker and the AMQP/DV API interfacing worked as expected. We do not see the HPC pedigree as a barrier to adoption for open source software leveraging the platform, but as a condition that must be managed for any continued efforts.

3 Performance Analysis

The system under test looked at comparative tests leveraging FDR Infiniband and Data Vortex interconnects between nodes across a range of message sizes from 8 bytes to 128 kilobytes. The goal of the project is not to determine if any particular transport is better than another, but to analyze efficacy of an interconnect for general purpose messaging middleware. Both interconnects were able to sustain message rates per node much higher than those observed in the published “RabbitMQ 1M Messages/sec Record” held by Pivotal on Google Cloud Platform.[4]

[4] https://content.pivotal.io/blog/rabbitmq-hits-one-million-messages-per-second-on-google-compute-engine With 30 nodes in the cluster, each node sustained approximately 43,404 ingress messages/sec for the duration of the benchmark once steady state was obtained. Our RabbitMQ configuration mirrors many aspects of theirs (with an admittedly smaller number of nodes).

For this paper we will look at 8, 256, 2048, and 8192 byte messages. The entire dataset and analysis is available through the project jupyter notebook[5].

3.1 Comparison Details

One of the artifacts of the transport differences is the means by which messages are consumed. Since the implementation for Data Vortex is synchronous, the application must poll a buffer to retrieve messages. Threads pulling messages from the DV network convert chunked messages into single AMQP messages and push them onto a local socket for handling by RabbitMQ. This means that most poll intervals are empty and occasionally we see large bursts of activity as the receive buffer fills. Unfortunately, this means that a normal "messages/sec" plot does not show the actual behavior of the DV network. As a result, the plot of message rates is smoothed as a moving average to better show when the system is actually handling active messages and not polling an empty buffer[6].

The message rate plots for this discussion show calculations at the point where messages are removed from VIC SRAM registers and pushed into node memory buffers. As we moved through the testing and messages sizes increased, we encountered large bottlenecks in threads pushing messages into RabbitMQ through the local socket which resulted in buffer overruns between DV VIC and node memory. These IO waits between memory structures show marked performance degradation after the Data Vortex network, and so we are not including those timings in the interconnect comparisons below. We do illustrate the degradation in the next discussion.

3.2 General Observations

Multiprocessing python performs well over Infiniband transport, with our test harness routinely achieving sustained rates well above 60,000 messages/sec per node. This represents better than 27% improvement over the Pivotal record for 1M messages/second. We consider this world class performance and adequately represents high performance throughput for RabbitMQ. We were not able to achieve the same level of performance using the C AMQP API directly with threads and so we do not expect the same overall performance for our test harness versus the python Infiniband implementation. The C API can achieve higher performance using multiprocessing, but the Data Vortex API does not support multiprocessing access, so a threaded model was all that was available to us for these tests.

[5] http://providentiaworldwide.com/projects/DV/notebooks/DataVortex_Analysis.ipynb

[6] Better accounting to show "traditional" views of message rates is a topic we have brought to the Data Vortex team, but is not available for this paper.

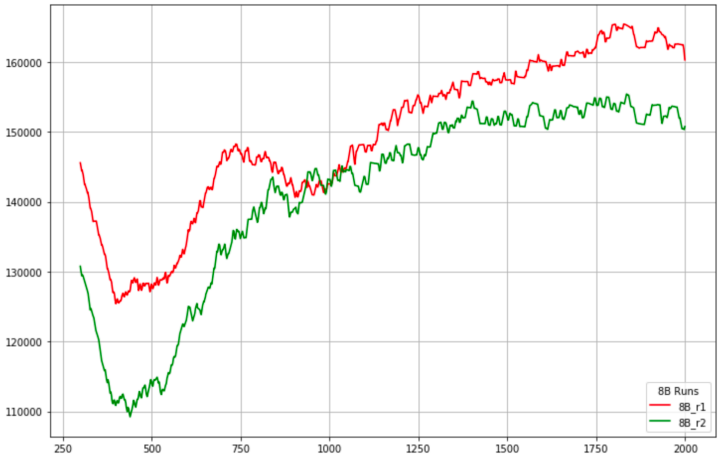
That said, the Data Vortex network performs very well for message delivery between nodes and was never a bottleneck in our testing. The comparisons below call out specifics as we move through message sizes[7].

3.3 8 Byte Performance

Data Vortex, with nearly all samples over 800,000 messages/sec performs more than 5 times better than the Infiniband tests.

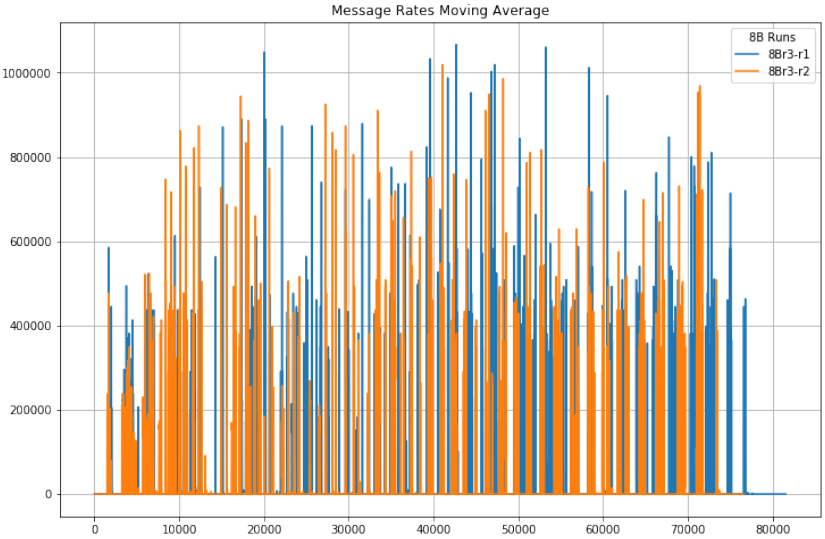
3.3.1 8B FDR Infiniband

	8B_r1	8B_r2
count	2000.000000	2000.000000
mean	150302.379370	142114.492697
std	38069.489926	43043.187694
min	14533.838700	15515.433300
25%	115394.525975	102502.065100
50%	172503.215150	168669.050050
75%	175716.244850	176793.805300
90%	178048.994230	179369.455090
95%	181190.053215	180730.108740
max	240024.643700	256103.299200



3.3.2 8B Data Vortex

	8Br3-r1	8Br3-r2
count	81,650.0000	76,708.0000
mean	24,367.1122	25,899.9608
std	155,509.3793	153,395.9505
min	0.0000	0.0000
25%	0.0000	0.0000
50%	0.0000	0.0000
75%	0.0000	0.0000
90%	0.0000	0.0000
95%	0.0000	0.0000
99%	836,636.6556	830,419.9375
max	3,399,120.2500	3,263,747.2500



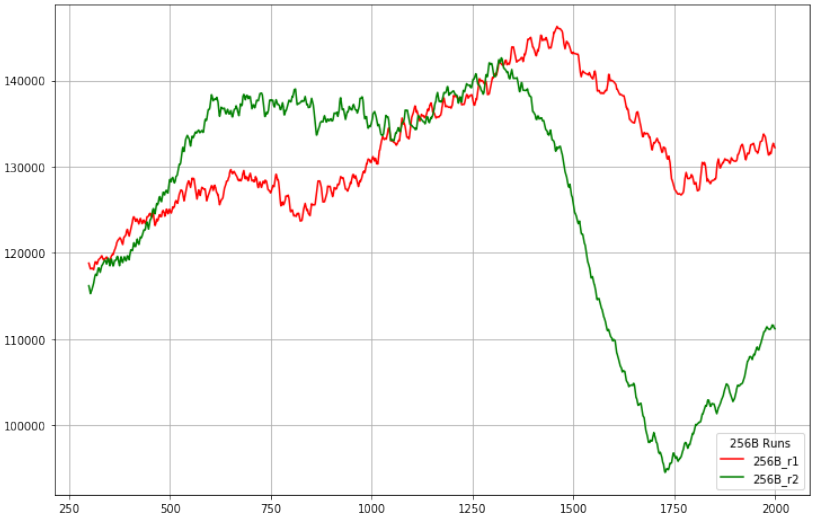
[7] The Jupyter notebook has results for all of the tested message sizes and links to github for the configuration details of RabbitMQ and all software developed for the tests.

3.4 256 Byte Performance

FDR Infiniband sustains performance of about 130,000 messages/sec across all runs, with the majority of samples above 200,000 for Data Vortex. Where the DV requires chunking messages into 64 bit words, this means that 32 messages are transferred for any single 256 byte message in the python implementation. Performance is still nearly twice that of the Infiniband implementation.

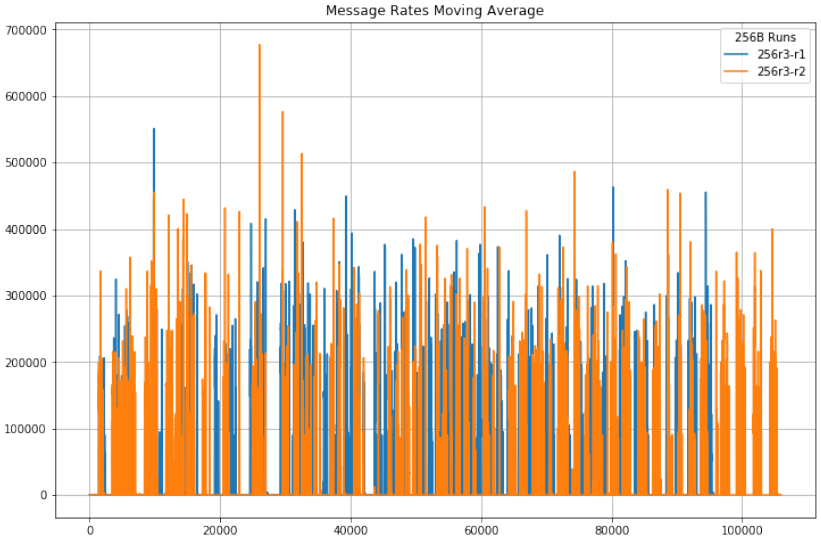
3.4.1 256B FDR Infiniband

	256B_r1	256B_r2
count	2000.0000	2000.0000
mean	130873.7446	123974.1974
std	38062.4215	38308.4512
min	11994.6701	21789.2178
25%	101817.4144	95694.5835
50%	144048.6414	125226.6518
75%	161141.4219	158591.8685
90%	169812.8229	171747.5988
95%	175182.9529	176608.2929
max	224189.9958	258566.8270



3.4.2 256B Data Vortex

	256r3-r1	256r3-r2
count	96,028.0000	106,129.0000
mean	20,719.4136	18,782.3233
std	102,774.0353	98,317.3925
min	0.0000	0.0000
25%	0.0000	0.0000
50%	0.0000	0.0000
75%	0.0000	0.0000
90%	0.0000	0.0000
95%	135,972.8125	60,707.8594
99%	583,683.2500	579,628.0800
max	1,712,257.5000	1,567,886.3750



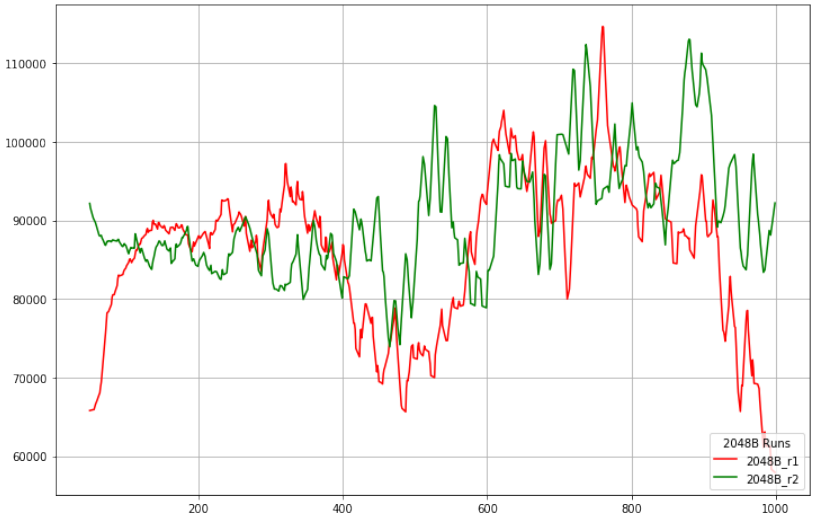
3.5 2048 Byte Performance

Data Vortex performance continues to average twice that for InfiniBand. 2K messages require 256 words for each single message in the traditional implementation. The affinity in the DV network for small, efficient messages bears out clearly as message size increases, though the

memory requirements for chunk reassembly increase dramatically when compared with the FDR test harness. As tests continued, memory management on the SUT became more difficult – with overruns resulting in failed test iterations and buffer size adjustments.

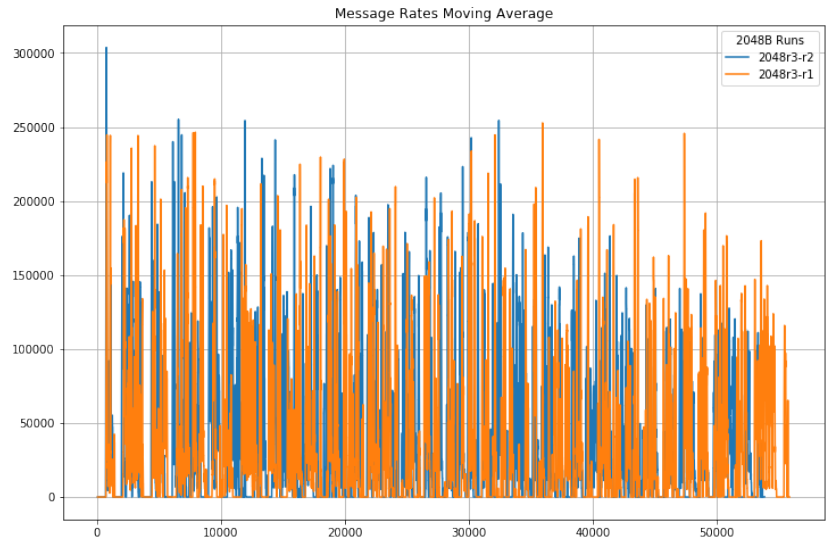
3.5.1 2K FDR Infiniband

	2048B_r1	2048B_r2
count	1000.0000	1000.0000
mean	84446.1632	90501.2204
std	32843.3797	32728.3918
min	18130.2160	26606.6628
25%	66163.3418	71262.9431
50%	82752.3570	86536.9374
75%	101637.6993	121854.8697
90%	128775.6994	131202.4679
95%	138156.1067	139758.6745
max	209783.3711	186020.8514



3.5.2 2K Data Vortex

	2048r3-r2	2048r3-r1
count	53,896.0000	55,915.0000
mean	36,984.6020	35,620.7460
std	99,616.3588	97,570.9014
min	0.0000	0.0000
25%	0.0000	0.0000
50%	0.0000	0.0000
75%	0.0000	0.0000
90%	174,365.1250	165,166.9688
95%	308,938.2188	307,338.5312
99%	431,723.6547	428,514.3125
max	556,288.7500	588,282.3125



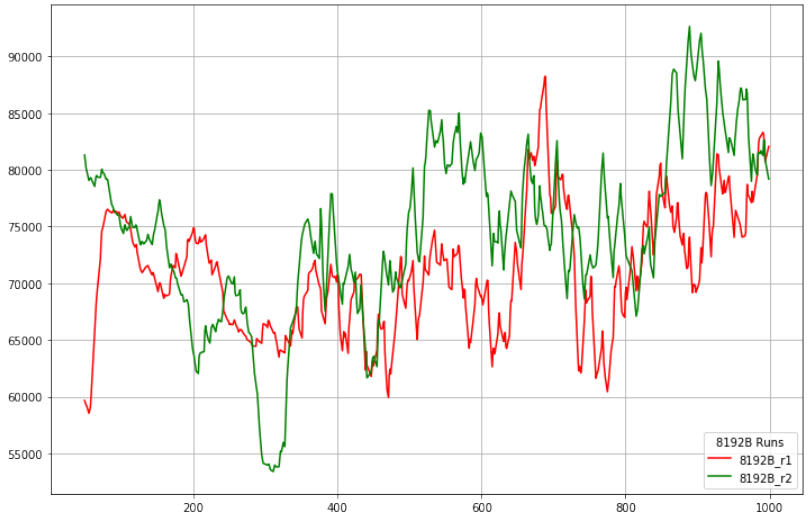
3.6 8192 Byte Performance

While FDR continues to perform very well, the Data Vortex also continues to maintain nearly twice the performance. Memory buffer overruns continued to be problematic during our test

runs and we exposed an anomaly in the DV API where it sometimes required additional polling loops to retrieve all chunks successfully[8].

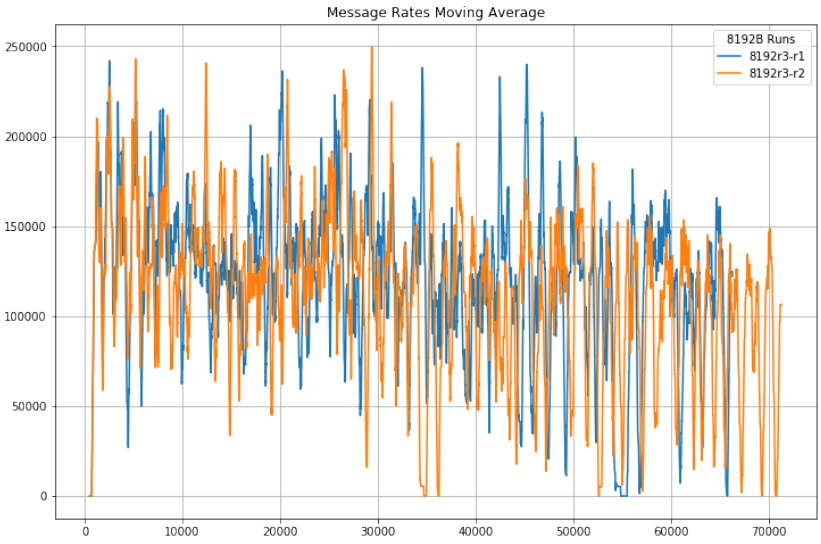
3.6.1 8K FDR Infiniband

	8192B_r1	8192B_r2
count	1000.000000	1000.000000
mean	70737.468268	74661.050176
std	26947.435333	25466.212653
min	21328.030500	23717.439400
25%	53330.657075	58020.973275
50%	68147.482150	71183.362800
75%	83160.400400	88446.366775
90%	107865.824970	107283.298190
95%	117244.185470	120536.790645
max	195842.565500	191430.242600



3.6.2 8K Data Vortex

	8192r3-r1	8192r3-r2
count	66,370.0000	71,397.0000
mean	120,148.7925	111,742.8893
std	152,927.3053	149,788.7659
min	0.0000	0.0000
25%	0.0000	0.0000
50%	0.0000	0.0000
75%	314,537.0938	304,339.1250
90%	337,532.5000	332,933.4062
95%	349,730.0625	346,330.7188
99%	392,121.5625	388,946.2137
max	548,690.2500	549,090.1875



3.7 131072 Byte Performance

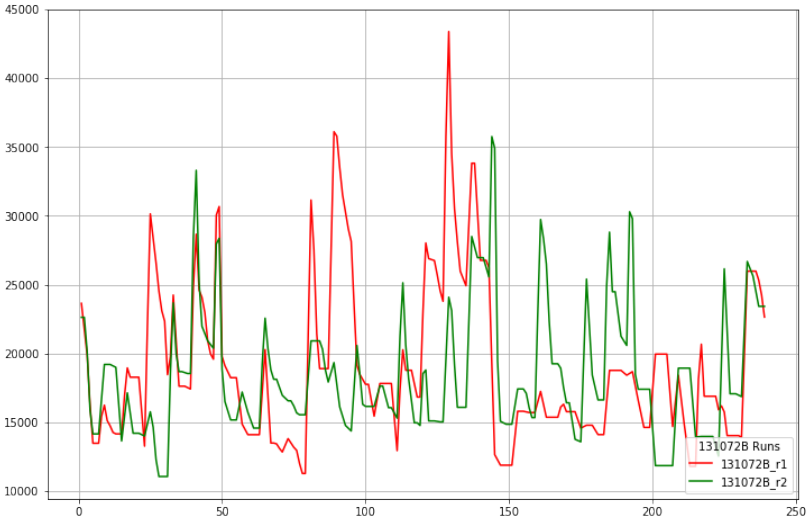
For 128K messages, we see the Data Vortex continuing to deliver linear performance against other message sizes. With 16,384 64-bit words per RabbitMQ message, the sustained performance of over 10x that of FDR shows the benefit of the topology. We do not understand

[8] While this behavior was not a show-stopper for our effort, it definitely made progress confusing as we were unsure if we were experiencing a bug from our memory management or unexpected behavior from the DV interaction. We are currently investigating the behavior with the Data Vortex team to see if there may be additional improvements from tuning in this area.

at this phase why the DV interconnect performance for 128K improves dramatically over the previous runs, but the values shown are consistent across 8 runs. However, due to memory constraints, each run could move only 600,000 messages. We suspect that if we were able to move millions of messages in a single run, we might see a natural flattening with the longer run rate. We cannot test this however, without additional memory in the cluster nodes.

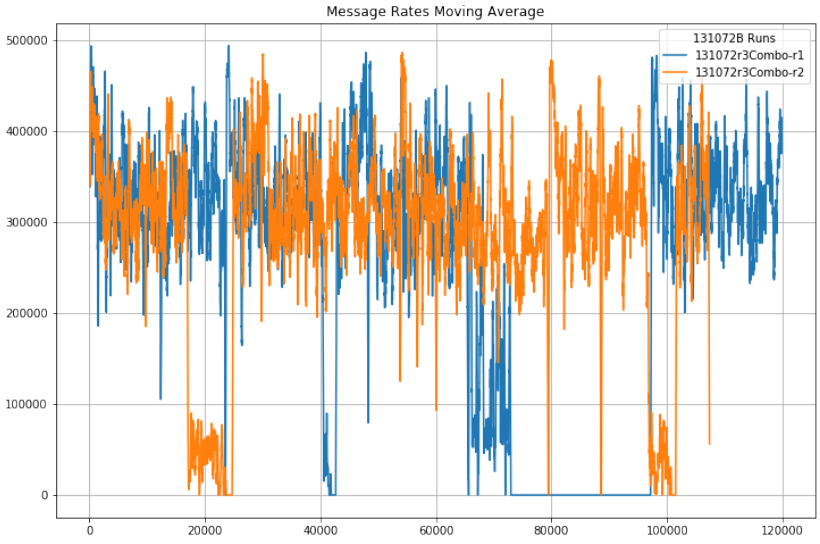
3.7.1 128K FDR Infiniband

	131072B_r1	131072B_r2
count	240.0000	240.0000
mean	19084.5645	18427.7446
std	6144.1322	5101.0509
min	11289.9506	11069.9820
25%	14636.1528	15077.1534
50%	17763.8247	17111.6836
75%	21665.9188	20585.3761
90%	26889.2560	25560.7995
95%	31163.7963	26972.3340
max	48466.3046	45946.5126



3.7.2 128K Data Vortex

	131072r3Combo-r1	131072r3Combo-r2
count	119,865.0000	107,411.0000
mean	245,338.6839	285,279.2535
std	220,357.6698	207,620.4934
min	0.0000	0.0000
25%	0.0000	0.0000
50%	289,742.0625	352,729.4688
75%	478,704.2500	491,301.7500
90%	503,899.2188	503,899.2188
95%	516,496.6875	516,496.6875
99%	554,289.1250	554,289.1250
max	3,161,967.5000	970,006.0000



3.8 AMQP vs DV Message Performance

As the tests progressed, it became clear that we were pushing against a bottleneck with regards to delivery of messages using the AMQP C API. The resulting behavior manifested in larger required buffer sizes to facilitate message reassembly and retention between the DV VIC SRAM registers and local node DRAM. Eventually the AMQP performance reached a crawl. For future work, investigating an alternative to TCP socket delivery mechanisms could prove invaluable for

overall performance improvements for distributed computing and streaming analytics applications.

The table below illustrates the relative performance between sending messages on the DV interconnect between physical nodes and AMQP through a local socket. We illustrate the 256B and 128K examples, but all data is available in the project notebook.

	256B Rabbit R1	256B Rabbit R2	256B Rabbit Avg	256B DV R1	256B DV R2	256B DV Avg	Avg Difference
Mean	135,902.82	18,782.31	60,587.88	159,968.00	119,976.01	139,972.00	79,384.12
Std Dev	102,772.37	98,316.09	60,587.88	99,809.56	94,851.23	97,330.40	36,742.51
	128K Rabbit R1	**128K Rabbit R2**	**128K Rabbit Avg**	**128K DV R1**	**128K DV R2**	**128K DV Avg**	**Avg Difference**
Mean	3,599.28	3,799.24	3,699.26	278,500.82	284,590.99	281,545.90	277,846.64
Std Dev	1,732.26	1,633.55	1,682.90	213,264.60	207,119.34	210,191.97	208,509.07

Table 2: AMQP vs DV Message Send Rates

4 General Analysis and Conclusions

The Data Vortex interconnect topology delivers on its claims of uniform scalability and predictable message behavior for small packets. Likewise, the platform API is well-documented and integrates easily into traditional HPC parallel computing paradigms and design patterns. The interconnect delivers far above the capabilities of current open source AMQP implementations on the same hardware – even when optimized for parallel task management and resources.

However, the same legacy in parallel computing presents challenges for adopting a variable length, loosely coupled architecture for general purpose, stream analytics, and big data processing pipelines and workflows. To see real improvements in open source messaging middleware, the right approach would be to develop native APIs for functional and higher-level languages as a means of wider adoption for the Data Vortex VIC.

4.1 Continued Efforts

Messaging middleware underlies most of the modern platform infrastructure, container architectures, and distributed computing platforms in the world. It has only occasionally been the recipient of optimizations despite its 40 year history, and rarely seen hardware optimizations tuned directly to messaging needs. Implementing a native API to solve messaging middleware problems and to migrate messaging patterns and primitives into the network could dramatically improve performance and reduce operational expenses for traditional messaging systems. We recommend continued work in this and similar projects to investigate truly asynchronous and asymmetric messaging atop the Data Vortex network.

5 References

RabbitMQ C AMQP Client Library. https://github.com/alanxz/rabbitmq-c (cloned from source on 10 January 2018.

AMQP Specification. http://www.amqp.org/resources/download.

RabbitMQ Docker Container. Maintained by Docker Community and Pivotal Software. https://hub.docker.com/r/_/rabbitmq/ (cloned from source on 10 January 2018).

Vortex Topology Messaging Project Notebook. Jupyter Notebook: http://providentiaworldwide.com/projects/DataVortex/notebooks/DataVortexAnalysis.ipynb

Data Vortex C API. http://www.datavortex.com/programming/

Logan, Martin; Eric Merritt, Richard Carlsson. *Erlang and OTP in Action*. Manning Publications, 2011.

Roy, Gavin M. *RabbitMQ in Depth*. Manning Publications, 2018.

Nickoloff, Jeff. *Docker in Action*. Manning Publications, 2016.