Aeron®

AWS Performance Testing

Step-by-Step guide to benchmarking Aeron in your AWS environment.

aeron.io

Table of Contents

01 Introduction

The Aeron team at Adaptive and Amazon Web Services (AWS) have published Aeron benchmark results to demonstrate the performance that can be achieved running Aeron on AWS.

This paper provides technical instructions on how to set up and run performance tests with Aeron. It also includes detailed results from our own performance tests, which can give you an idea of what to expect from your own testing.

Detailed latency and throughput results are below. In summary, Aeron Premium is almost 500 times faster at the 99th percentile than other commonly used messaging protocols for data transport and almost 60 times faster for end-to-end encrypted transport than other commonly used encryption protocols. For clustered state replication, Aeron Premium halves latency while achieving up to 8x throughput compared to Aeron Open Source.

02 What is Aeron?

Aeron is the cloud-native, open-source, low-latency message transport and cluster technology developed by Adaptive and used by financial services firms globally to build sophisticated high-performance trading systems. Adaptive has worked with AWS since 2014, using its cloud-based technology to run Aeron to deploy purpose-built trading solutions for financial institutions.

Aeron consists of Aeron Transport for messaging, and Aeron Cluster for sequenced, persisted, state replication. Aeron Premium, from Adaptive, provides an additional set of components to enhance performance, security and resilience.

03 Why Aeron?

Aeron Transport and Aeron Cluster solve two key pain points for Capital Markets systems in the cloud.

1. Performance: Low-latency, high-throughput data transport, and dissemination

2. High-availability: 24/7 ‘always on’ systems which are fault tolerant

A. Aeron Transport: Data Transport and Reliable Dissemination

The Challenges:

Reliably moving data at predictable, low latencies is fundamental to any system involved in low-latency, high-frequency markets. Data needs to be moved from one machine to many others - market data distribution is a common example of this. Hardware-based multicast solutions solved this problem for capital markets businesses before the cloud. Multicast in the cloud requires, unfortunately, a compromise in performance.

The Solution:

Aeron Transport is able to reliably and predictably transport data across IPC (inter-process communication) and local and wide area networks. It adds minimal overhead to the latency of the underlying network, while providing flow and congestion control built for today's multi-tenant high capacity networks.

Aeron Transport is especially well-suited to message transport in the cloud, with features such as:

- **UDP-based messaging** with capital markets-tuned flow and congestion-control algorithms.
- **Multi-destination cast**, which provides a high throughput multicast-like pattern for the cloud.
- **DPDK kernel bypass**, which provides blazing fast network throughput rates and low latencies, by directly accessing the underlying physical network card.
- **Natural batching**, which enables asynchronous message passing to reach incredibly high throughput.

These features come together to allow Aeron Transport to transmit data reliably and at extremely fast rates on AWS.

For durable messaging, Aeron Archive records, writes and replays to storage. This enables users to create fast, complex messaging topologies tuned for their requirements, including large-scale, reliable, market data distribution.

B. Aeron Cluster: Sequenced, persisted, and highly-available, in the cloud

The Challenges:

Traditional cloud approaches to resilience require a scale-out approach, with architecture that sacrifices consistency for availability. Capital markets systems require consistency, performance, and sequencing, often running entire markets on one or two machines to achieve it. Incredibly short Recovery Time Objectives (RTO) and Recovery Point Objectives (RPO) in the order of milliseconds are common.

Recovering from system outages where data consistency or the ordering of transactions is required entails extremely complex reconciliations and, ultimately, some of the costliest payouts in compensation to impacted customers.

The cloud engineering paradigm, while offering fast provisioning and scalability, is also one of ephemerality - network interfaces are patched, processes are migrated between machines, and local storage can disappear. Solving these engineering requirements in the cloud has been thought to be near impossible.

The Solution:

Aeron Cluster is uniquely well-suited for this paradigm. It provides developers with a resilient, performant platform that can process over 2 million messages per second with a 99th percentile latency of 103 microseconds (detailed results are in the following section).

When underlying cloud services are restarted or migrated, Aeron Cluster seamlessly continues service operation with recovery in milliseconds.

This architecture enables developers to build highly-available, resilient systems with a minimum of infrastructure and yet still achieve incredible throughput and performance. Developers concentrate on the logic of their business domain, relying on the resilience guarantees provided by Aeron Cluster for their RTO and RPO needs.

04 Benchmarking Aeron

Aeron on AWS delivers exceptional throughput and low latency. We've open-sourced the source code of the benchmarks and published a guide so you can replicate them in your own environment.

Testing took place with a range of AWS services, including Amazon Elastic Compute Cloud (EC2) instances, Amazon Elastic Block Store (EBS) volumes, and Amazon Elastic Compute Cloud (EC2) Placement Groups.

To start, we tested Aeron at its lowest level, using Aeron Transport to push network throughput and latencies to their extreme. We also tested Aeron Transport Security (ATS), an Aeron Premium component that uses industry-standard cryptography primitives. We then tested Aeron Cluster, to show the throughput and latency that a highly available system could achieve built on top of it.

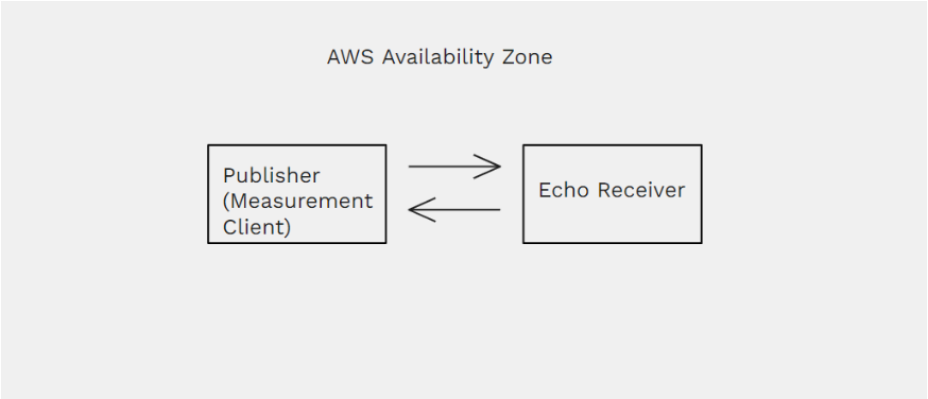
We ran tests for both open-source Aeron and Aeron Premium. These differ only in how they access the network card: Open-source Aeron uses BSD sockets whilst Aeron Premium uses DPDK. Aeron DPDK kernel bypass allows applications to directly access network interfaces and hardware resources (AWS Nitro instances make the underlying hardware available), reducing the overhead associated with traditional kernel-based networking and thus dramatically improving messaging latency and increasing throughput.

A. Aeron Transport - The latency of a round trip without persistence

i. Test Setup

We used Aeron Transport to benchmark the underlying latency and throughput when sending messages across the AWS network. We used Amazon EC2 Cluster Placement Groups to control the proximity of AWS instances within an AWS Availability Zone.

Figure: Aeron Transport Test Set-up in an AWS Availability Zone



To measure latency, we performed five test runs of our test case: an echo test of 100,000 288-byte messages per second.

For throughput, we wanted to understand the maximum while still meeting a given latency budget within an Availability Zone. We chose a latency ceiling of 1 millisecond at the 99th percentile, stopping at throughputs that gave results above this threshold.

1. Technical Set-Up - Prerequisites

- Some provisioned VMs or machines (Get in touch if you would like to use our Aeron benchmark infrastructure provisioning modules to help with setting this up.)
- Basic level of Linux & networking knowledge

2. Technical Set-up Instructions

Benchmark scripts

Running Aeron Echo benchmarks requires two virtual machines to be provisioned. For machine setup please follow the OS setup guide.

To get started with the Aeron benchmarks do the following:

- Clone the benchmarks project:
  `$ git clone https://github.com/real-logic/benchmarks aeron-benchmarks`
- Build the deployment package:
  `$ cd aeron-benchmarks`
  `$ ./gradlew clean deployTar`
  This command will generate an archive file build/distributions/benchmarks.tar.
  This step requires JDK 8+ to be installed on the local machine. It will then download Gradle and all of the dependencies.
- Copy the `build/distributions/benchmarks.tar file` to the test machines. Typically done with the scp command, e.g.:
  `$ scp build/distributions/benchmarks.tar remote-user@remote-machine:~`
- On each remote machine unpack the deployment archive.
  `$ tar xf benchmarks.tar -C <destination_dir>`
- On the local machine in the benchmarks project (i.e. in the `` `aeron-benchmarks/scripts` directory``) create a new wrapper script that will run the remote benchmarks of your choice.
  For example, create a file named test-aeron-echo.sh in the scripts directory and add the following contents into it. NB: All the values in angle brackets (<...>) will have to be replaced with the actual values.

```
# SSH connection properties
export SSH_CLIENT_USER=<SSH client machine user>
export SSH_CLIENT_KEY_FILE=<private SSH key to connect to the client machine>
export SSH_CLIENT_NODE=<IP of the client machine>
export SSH_SERVER_USER=<SSH server machine user>
export SSH_SERVER_KEY_FILE=<private SSH key to connect to the server machine>
export SSH_SERVER_NODE=<IP of the server machine>


# Set of required configuration options
export CLIENT_BENCHMARKS_PATH=<directory containing the unpacked benchmarks.tar>
export CLIENT_JAVA_HOME=<path to JAVA_HOME (JDK 8+)>
export CLIENT_DRIVER_CONDUCTOR_CPU_CORE=<CPU core to pin the ‘conductor’ thread>
export CLIENT_DRIVER_SENDER_CPU_CORE=<CPU core to pin the ‘sender’ thread>
export CLIENT_DRIVER_RECEIVER_CPU_CORE=<CPU core to pin the ‘receiver’ thread>
export CLIENT_LOAD_TEST_RIG_MAIN_CPU_CORE=<CPU core to pin ‘load-test-rig’ thread>
export CLIENT_NON_ISOLATED_CPU_CORES=<a set of non-isolated CPU cores>
export CLIENT_CPU_NODE=<CPU node (socket) to run the client processes on>
export CLIENT_AERON_DPDK_GATEWAY_IPV4_ADDRESS=
export CLIENT_AERON_DPDK_LOCAL_IPV4_ADDRESS=
export     CLIENT_SOURCE_CHANNEL=”aeron:udp?endpoint=<SOURCE_
IP>:13100|interface=<SOURCE_IP>/24”
export
CLIENT_DESTINATION_CHANNEL=”aeron:udp?endpoint=<DESTINATION_
IP>:13000|interface=<DESTINATION_IP>/24”
export SERVER_BENCHMARKS_PATH=<directory containing the unpacked benchmarks.tar>
export SERVER_JAVA_HOME=<path to JAVA_HOME (JDK 8+)>
export SERVER_DRIVER_CONDUCTOR_CPU_CORE=<CPU core to pin the ‘conductor’ thread>
export SERVER_DRIVER_SENDER_CPU_CORE=<CPU core to pin the ‘sender’ thread>
export SERVER_DRIVER_RECEIVER_CPU_CORE=<CPU core to pin the ‘receiver’ thread>
export SERVER_ECHO_CPU_CORE=<CPU core to pin ‘echo’ thread>
export SERVER_NON_ISOLATED_CPU_CORES=<a set of non-isolated CPU cores>
export SERVER_CPU_NODE=<CPU node (socket) to run the server processes on>
export SERVER_AERON_DPDK_GATEWAY_IPV4_ADDRESS=
```

```
export SERVER_AERON_DPDK_LOCAL_IPV4_ADDRESS=
export SERVER_SOURCE_CHANNEL="${CLIENT_SOURCE_CHANNEL}"
export SERVER_DESTINATION_CHANNEL="${CLIENT_DESTINATION_CHANNEL}"

# (Optional) Overrides for the runner configuration options
#export MESSAGE_LENGTH="288" # defaults to "32,288,1344"
#export MESSAGE_RATE="100K"  # defaults to "1M,500K,100K"


# Invoke the actual script and optionally configure specific parameters
"aeron/remote-echo-benchmarks" --client-drivers "java" --server-drivers "java" --mtu
8K --context "my-test
```

- Run the wrapper script created in the previous step.
  `$ ./test-aeron-echo`

OS setup

We have used an Ubuntu 22.04 for running Aeron benchmarks as this was the latest Ubuntu LTS at the time of the test and to have access to the kernel bypass networking (DPDK).

Tools

Aeron benchmark scripts require several tools to be installed:

- `numactl`
- `jq`
- `lstopo`
- `lsb_release`

On Ubuntu these can be installed with a single command:
`$ sudo apt-get install numactl jq hwloc`

Kernel configuration parameters

What's required to get a stock Linux image configured to optimal latency? - This is the topic that is well covered by external resources such as:

- The Black Magic of Systematically Reducing Linux OS Jitter by Gil Tene.
- Low latency tuning guide by Eric Rigtorp.

Here are the specific parameters that we have tuned when running the benchmarks. Usually we create a file `/etc/sysctl.d/99-aeron-benchmarks.conf` with the following contents:

```
# Virtual memory
vm.swappiness = 0
vm.stat_interval=120

# Memory
vm.min_free_kbytes = 8388608
vm.zone_reclaim_mode = 0
kernel.numa_balancing = 0

# Networking
net.core.rmem_max = 16777216
net.core.wmem_max = 16777216
net.ipv4.tcp_rmem = 4096 2097152 16777216
net.ipv4.tcp_wmem = 4096 2097152 16777216
```

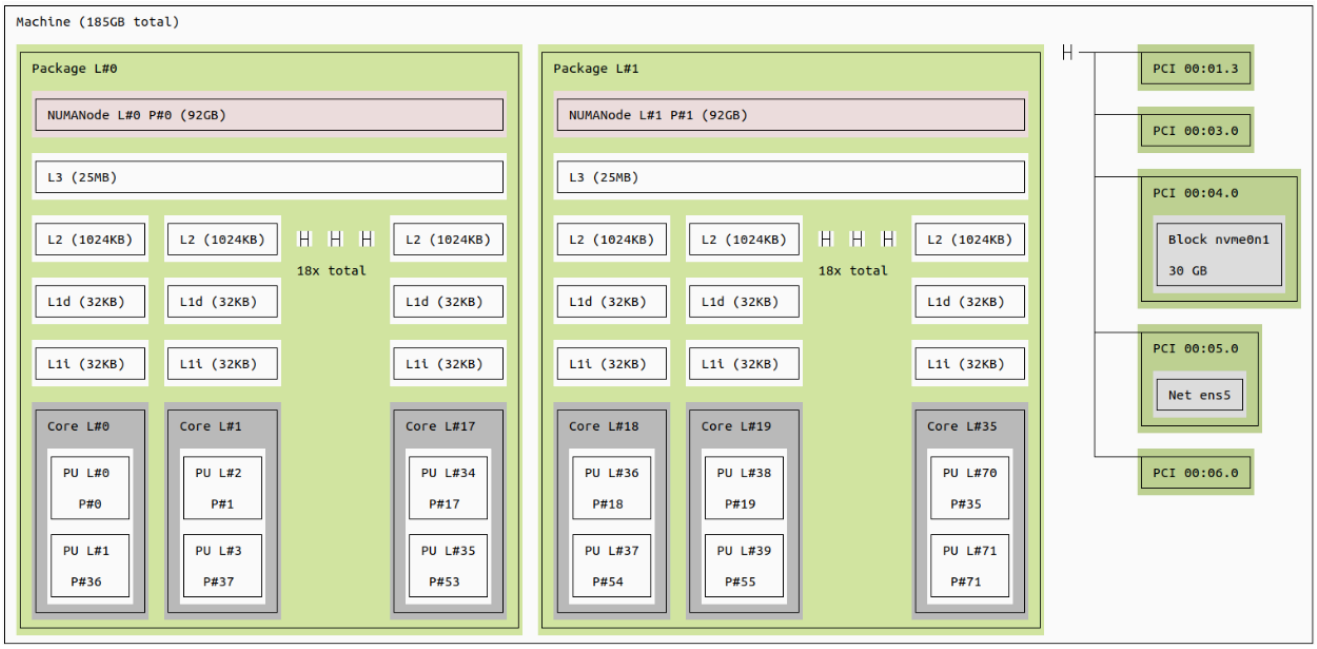
```
net.ipv4.tcp_max_syn_backlog = 8192
net.ipv4.tcp_slow_start_after_idle = 0
net.ipv4.tcp_syn_retries = 2
net.ipv4.tcp_tw_reuse = 1

# File System
vm.dirty_ratio = 80
vm.dirty_background_ratio = 5
vm.dirty_expire_centisecs = 12000

# perf (profiling)
kernel.perf_event_paranoid = -1
kernel.kptr_restrict = 0
kernel.perf_event_max_stack = 1024
kernel.perf_event_mlock_kb = 8096
```

CPU core isolation

Aeron benchmark scripts pin threads to the dedicated CPU cores. To ensure that nothing else is running on those cores it is necessary to isolate them.
To find the cores to isolate, do the following:

- Determine to which CPU node (socket) the network cards are attached.
  This can be done by running the `lstopo-no-graphics` (or `lstopo`) command which lists the PCI bridges after the cores of a CPU package.

  For example here is an output from the lstopo command:



Here the NICs are attached to the second socket.

- Choose the cores to isolate.

  Use the CPU socket from the previous step and select a set of cores to be isolated. Aeron benchmarks will use up to 8 CPU cores to pin the threads so at least 8 physical CPUs cores should be isolated.

  You can use the lscpu utility to find the mapping of the logical cores to the physical cores and sockets, e.g.:

```
$ lscpu -e
CPU NODE   SOCKET CORE   L1d:L1i:L2:L3       ONLINE
  0        0      0      0      0:0:0:0             yes
  1        0      0      1      1:1:1:0             yes
  2        0      0      2      2:2:2:0             yes
  3        0      0      3      3:3:3:0             yes
  4        0      0      4      4:4:4:0             yes
  5        0      0      5      5:5:5:0             yes
  6        0      0      6      6:6:6:0             yes
  7        0      0      7      7:7:7:0             yes
  8        0      0      8      8:8:8:0             yes
  9        0      0      9      9:9:9:0             yes
 10        0      0      10     10:10:10:0          yes
 11        0      0      11     11:11:11:0          yes
 12        0      0      12     12:12:12:0          yes
 13        0      0      13     13:13:13:0          yes
 14        0      0      14     14:14:14:0          yes
 15        0      0      15     15:15:15:0          yes
 16        0      0      16     16:16:16:0          yes
 17        0      0      17     17:17:17:0          yes
 18        1      1      18     32:32:32:1          yes
 19        1      1      19     33:33:33:1          yes
 20        1      1      20     34:34:34:1          yes
 21        1      1      21     35:35:35:1          yes
 22        1      1      22     36:36:36:1          yes
 23        1      1      23     37:37:37:1          yes
 24        1      1      24     38:38:38:1          yes
 25        1      1      25     39:39:39:1          yes
 26        1      1      26     40:40:40:1          yes
 27        1      1      27     41:41:41:1          yes
 28        1      1      28     42:42:42:1          yes
 29        1      1      29     43:43:43:1          yes
 30        1      1      30     44:44:44:1          yes
 31        1      1      31     45:45:45:1          yes
 32        1      1      32     46:46:46:1          yes
 33        1      1      33     47:47:47:1          yes
 34        1      1      34     48:48:48:1          yes
 35        1      1      35     49:49:49:1          yes
 36        0      0      0      0:0:0:0             yes
 37        0      0      1      1:1:1:0             yes
 38        0      0      2      2:2:2:0             yes
 39        0      0      3      3:3:3:0             yes
 40        0      0      4      4:4:4:0             yes
 41        0      0      5      5:5:5:0             yes
 42        0      0      6      6:6:6:0             yes
 43        0      0      7      7:7:7:0             yes
 44        0      0      8      8:8:8:0             yes
 45        0      0      9      9:9:9:0             yes
 46        0      0      10     10:10:10:0          yes
 47        0      0      11     11:11:11:0          yes
 48        0      0      12     12:12:12:0          yes
 49        0      0      13     13:13:13:0          yes
 50        0      0      14     14:14:14:0          yes
 51        0      0      15     15:15:15:0          yes
 52        0      0      16     16:16:16:0          yes
 53        0      0      17     17:17:17:0          yes
 54        1      1      18     32:32:32:1          yes
 55        1      1      19     33:33:33:1          yes
 56        1      1      20     34:34:34:1          yes
 57        1      1      21     35:35:35:1          yes
 58        1      1      22     36:36:36:1          yes
 59        1      1      23     37:37:37:1          yes
 60        1      1      24     38:38:38:1          yes
 61        1      1      25     39:39:39:1          yes
```

```
62        1      1      26     40:40:40:1          yes
63        1      1      27     41:41:41:1          yes
64        1      1      28     42:42:42:1          yes
65        1      1      29     43:43:43:1          yes
66        1      1      30     44:44:44:1          yes
67        1      1      31     45:45:45:1          yes
68        1      1      32     46:46:46:1          yes
69        1      1      33     47:47:47:1          yes
70        1      1      34     48:48:48:1          yes
71        1      1      35     49:49:49:1          yes
```

Given the above output the correct list of cores to isolate would be 18-25,54-61, because we want to isolate 8 physical cores and thus must include both hyper-threads of each core in the isolation list.

- Isolate the cores by adding the corresponding boot options.
  We then need to add these cores to the OS boot parameters (e.g. by editing the `/etc/default/grub file`):

  ```
  GRUB_CMDLINE_LINUX="... isolcpus=18-25,54-61 nohz_full=18-25,54-61
  rcu_nocbs=18-25,54-61"
  ```

File system options

We have used an ext4 file system for configuring the storage devices (local SSDs, EBS, EFS discs etc.) with the following parameters:
`$ sudo mount -o defaults,noatime,nodiratime,discard,nobarrier ...`

The Aeron directory was placed under the `` `/dev/shm` `` which uses a tmpfs in-memory file system.

Key Aeron configuration options

Aeron scripts rely mostly on the low-latency-driver.properties and low-latency-archive.properties files for configuring the Media Driver and the Archive respectively. However there are several configuration options that must/can be set via the scripts as discussed below.

MTU

Aeron MTU can be configured at the media driver level (via the aeron.mtu.length and the `aeron.ipc.mtu.length` configuration options) or at the channel level via the mtu URI parameter. The MTU value controls the following:

- the size of the max network packet/IPC payload that the Aeron can send.
- the max number of bytes that can be claimed via the tryClaim API which is equal to the MTU minus the media message header, i.e. for Aeron Transport it is `` `mtu - 32` `` and for Cluster it is `` `mtu - 64` `` bytes. For example with the default MTU the max message size that can be sent via transport is 1374 bytes

Aeron MTU is set by default to 1408 bytes and is therefore tailored for the network MTU of 1500 bytes. The MTU value must be aligned by (be a multiple of) 32 bytes and must be less than the network MTU as it must accommodate for the network headers, i.e. the UDP header is 8 bytes, the IPv4 header is 20-60 bytes/IPv6 header is 40 bytes and the Ethernet header is 18 bytes which in total add up to 88 bytes of headers.

It is important to set Aeron MTU to match the network MTU for efficiency reasons. If the Aeron MTU is much smaller than the network MTU this will negatively affect the max throughput. And if it is larger then the Aeron packets will be split at the network level which will increase the negative effects of message loss.

For the jumbo frames where the NIC MTU is around 9000 set the Aeron MTU to 8KB, because setting it to a value larger than 8KB would break channels that use the 64KB term buffers (e.g. Archive control channel). As the term buffer length limits the max message size that can be published to 1/8 of the term buffer length (or 16MB whichever is smaller). So for the term buffer length of 64KB the max message size is therefore 8KB and the Aeron MTU for the channel must not exceed the max message size for that channel.

The Aeron benchmark scripts use --mtu parameter to override the Aeron MTU value.

File sync level

The file sync level must be set via the following configuration properties: `aeron.archive.file.sync.level and aeron.archive.catalog.file.sync.level`, where the Catalog file sync level must be equal or higher than the Archive file sync level.

This value defines how the data in the Archive is persisted to disc:

- **0** (default) - normal write is performed (pwrite on Linux), i.e. the data is written to the page cache and the OS is responsible for writing the dirty pages back to the disc asynchronously.
- **1** - normal write + sync file data (fdatasync on Linux), i.e. the data is written and synced to disc.
- **2** - normal write + sync file data and metadata (fsync on Linux), i.e. the data and the metadata is written and synced to disc.

The Aeron benchmark scripts use --fsync parameter to override the file sync level.

Send vector capacity (C media driver only)

When using the C media driver it is possible to specify the size of the send (`aeron.sender.io.vector.capacity`) and receive (`aeron.receiver.io.vector.capacity`) vectors as well the max number of the packets that the publication can send (`aeron.network.publication.max.messages.per.send`). They all default to two.

The Aeron benchmark scripts will set all of the three configuration properties to the same value as defined by the `AERON_NETWORK_PUBLICATION_MAX_MESSAGES_PER_SEND` environment variable. If it is not set then 2 is used as the default value. Increasing this default might help in the throughput tests.

ii. Test Results - Summary

The results of the testing were as follows:

- Latency of 66 microseconds, dropping to 32 microseconds with Aeron Premium kernel bypass* at 100k messages/second. This compares to 22,151 microseconds at 25k messages/second for other commonly used messaging protocols, Aeron is 500 times faster with 4 times the message volume.
- For encrypted transport, using Aeron Premium Transport Security (ATS) and kernel bypass, a latency of 46 microseconds* was measured. This compares to 384 microseconds and 2,699 microseconds for other commonly used encrypted protocols*. This represents an improvement of between 8 and 58-fold for Aeron Transport Security.
- Throughput of 350,000 messages/second with Aeron open source. With Aeron Premium, throughput leapt eight-fold, to over 3,000,000 messages/second.

** Apart from throughput results, all Aeron results quoted relate to a round trip time at the 99th percentile of a 288 byte message and a rate of 100,000 messages/second*

iii. Test Results - Detailed

The tables and charts below give a more detailed view of the results we achieved testing Aeron Transport. If you have questions regarding these, please get in touch.

Table: Aeron Transport Max Throughput (* 99th percentile less than 1ms on c5n.9xlarge)

	32 bytes	288 bytes	1344 bytes
Java	2.2M	400K	400K
C	1.5M	350K	300K
C with DPDK (Aeron Premium)	8M	3M	700K
C with ATS + DPDK (Aeron Premium)	8M	3M	700K
Aeron Premium ratio compared with C	5.3	8.6	2.3

Figure: Aeron Transport Max Throughput for a 32-byte message with 99th percentile less than 1 millisecond (throughput denoted in driver label in the key)

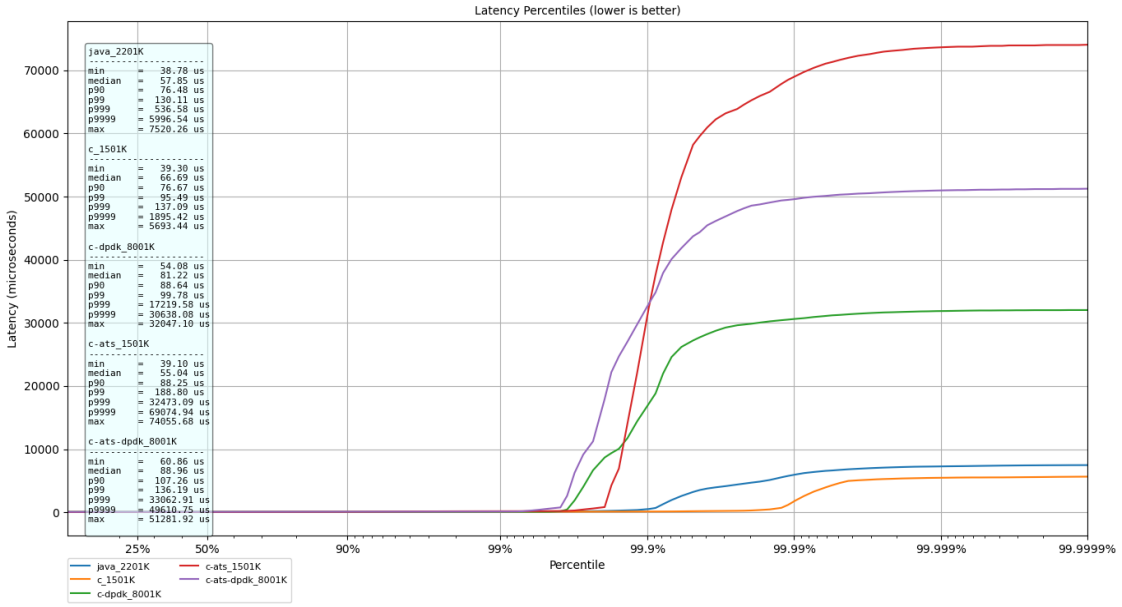


Figure: Aeron Transport Max Throughput for a 288-byte message with 99th percentile less than 1 millisecond (throughput denoted in driver label in the key)

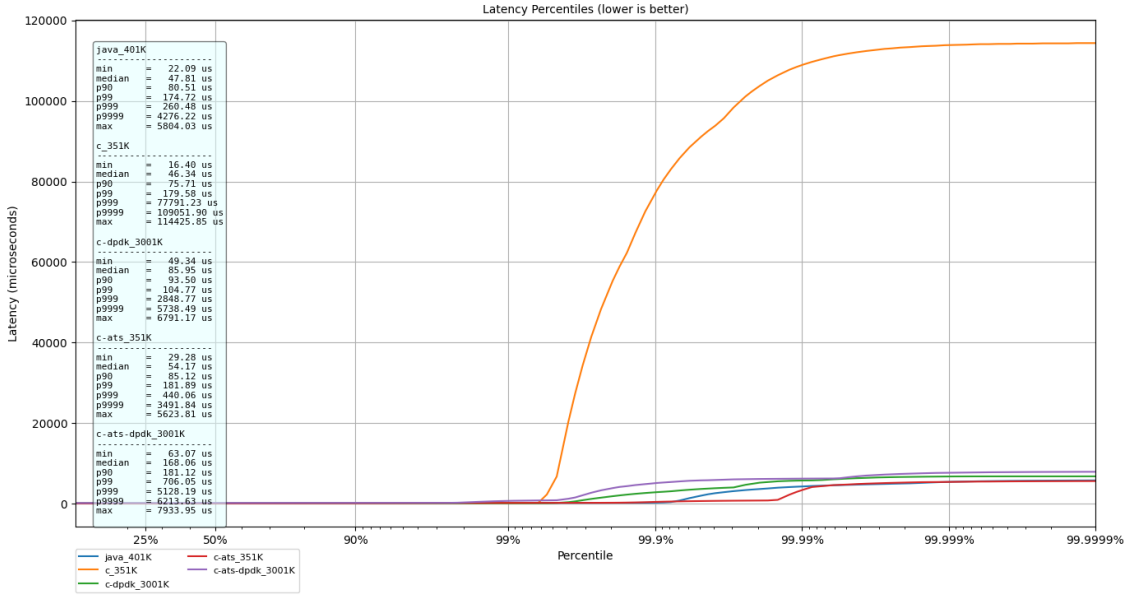


Figure: Aeron Transport Max Throughput for a 1344-byte message with 99th percentile less than 1 millisecond (throughput denoted in driver label in the key)

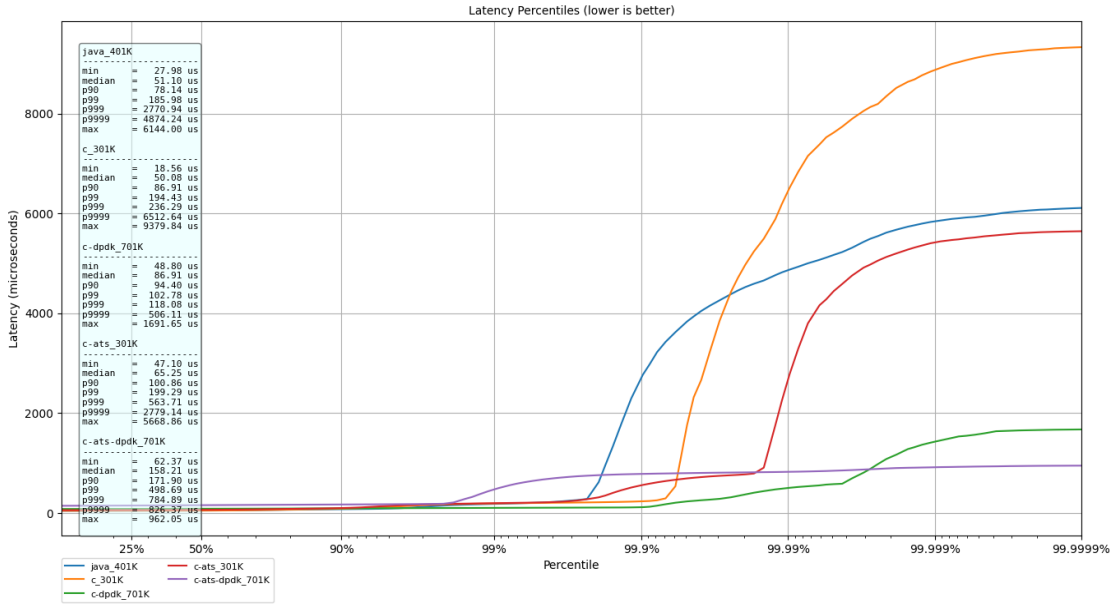


Table: Aeron Transport round trip latency @ 100,000 32 byte msg/sec (µs)

	P50	P99	P999	Max
Java	35	57	73	690
C	36	57	69	274
C-DPDK	24	32	49	354
Aeron Premium ratio compared with C	0.667	0.561	0.710	1.292

Figure: Aeron Transport round trip latency for a 32-byte message at 100,000 messages per second

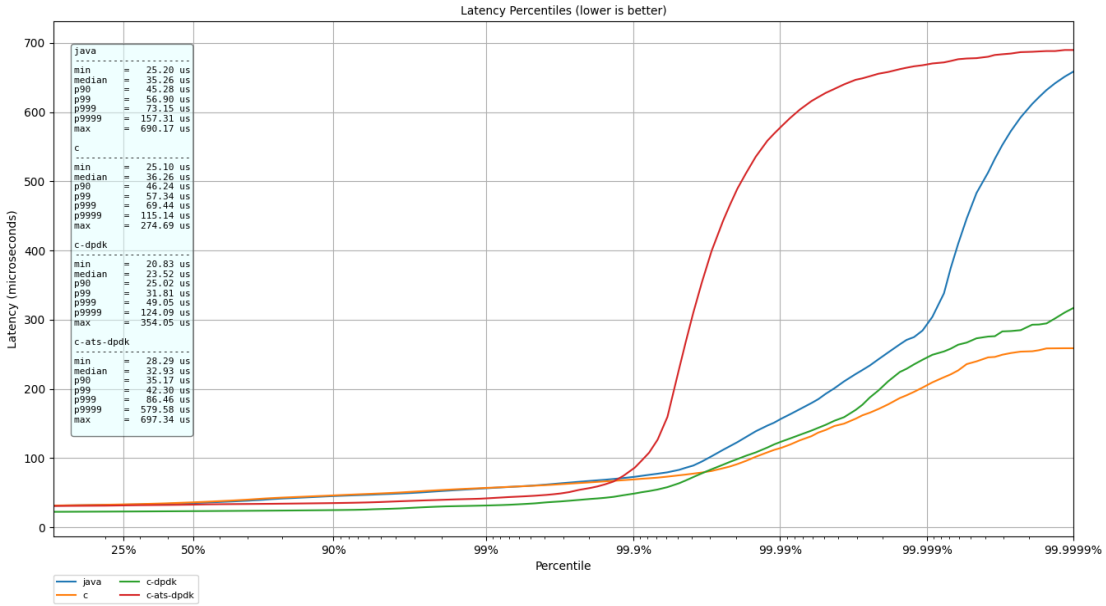


Table: Aeron Transport round trip latency @ 100,000 288 byte msg/sec (µs)

	P50	P99	P999	Max
Java	36	73	121	490
C	34	66	94	314
C-DPDK	36	43	47	283
Aeron Premium ratio compared with C	1.06	0.65	0.50	0.90

Figure: Aeron Transport round trip latency for a 288-byte message at 100,000 messages per second

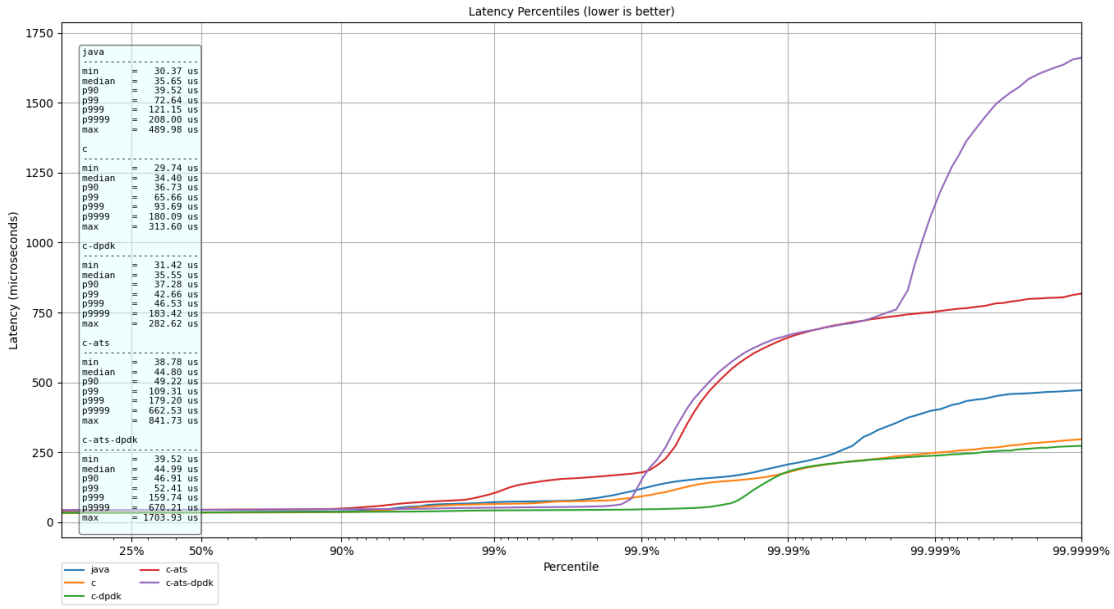
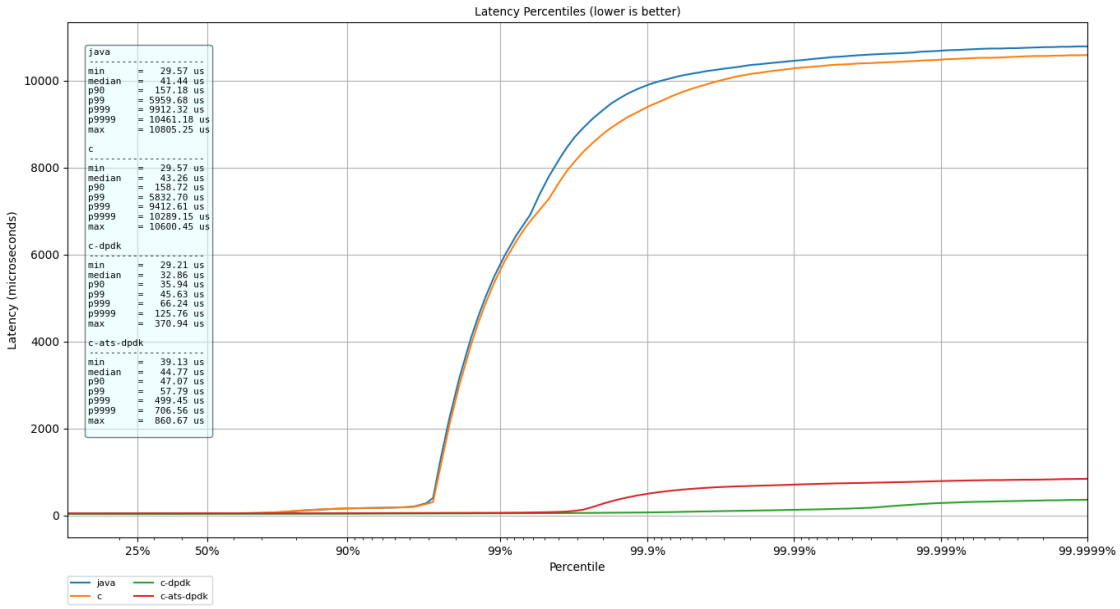


Table: Aeron Transport round trip latency @ 100,000 1344 byte msg/sec (µs)

	P50	P99	P999	Max
Java	41	5959	9912	10805
C	43	5832	9412	10600
C-DPDK	32	46	66	370
Aeron Premium ratio compared with C	0.744	0.008	0.007	0.035

Figure: Aeron Transport round trip latency for a 1344-byte message at 100,000 messages per second



B. Aeron Cluster - High-throughput and low-latency with high-availability

For more information on how Aeron Cluster works, please refer to this overview.

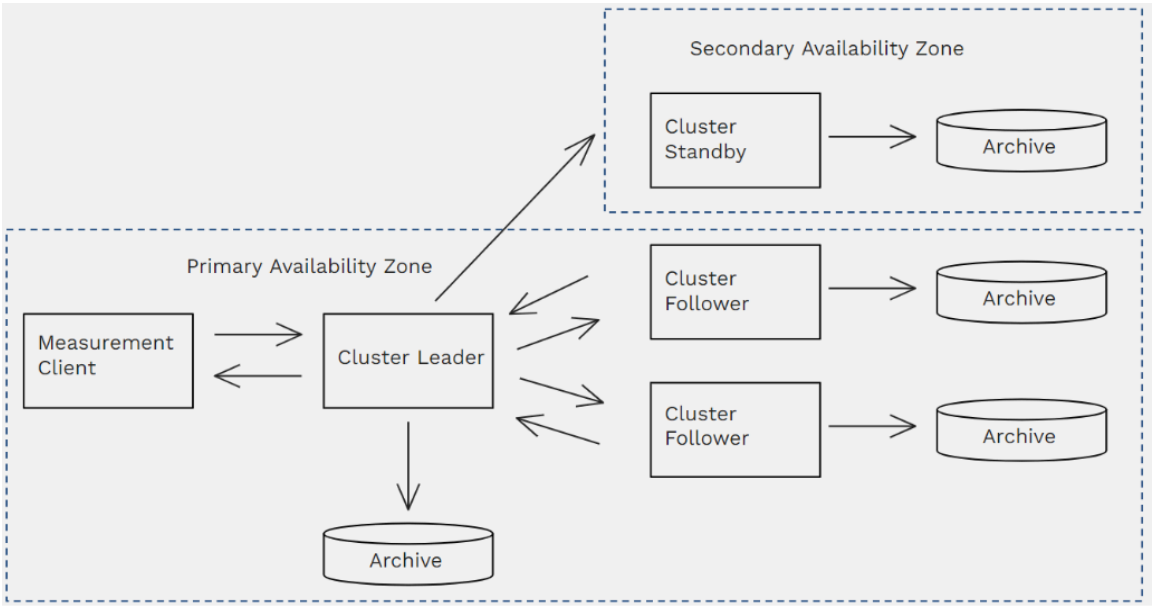
With Aeron Cluster, we want to benchmark the latency and throughput of a round-trip response where state is replicated across a three-node Aeron Cluster system. We've tested two different deployment configurations, the first is detailed below. For the second test set-up, please see the appendix.

i. Test Set-Up 1: Cluster Placement Group - a set-up optimized for performance

Here, we deployed Aeron Cluster nodes in the same AWS Availability Zone (AZ). This means that messages sent to the cluster are replicated to a quorum of other nodes within the same Availability Zone.

This configuration gives latency benefits but it comes with a redundancy trade-off when compared to deploying nodes across Availability Zones. When deployed across AZs, if the primary Availability Zone is lost, the system can be brought back up from the messages replicated to a secondary AZ through the use of Aeron Premium Cluster Warm Standby.

Figure: Aeron Cluster Test Set-up using AWS Cluster Placement Group and Aeron Cluster Standby in a Secondary Availability Zone.



As with the Aeron Transport tests, our Aeron Cluster testing covered latency and throughput.

For latency, we tested the performance of Aeron Cluster at 100,000 288-byte messages per second.

For throughput, we wanted to understand the maximum throughput while still meeting a given latency within an Availability Zone. We chose a latency ceiling of one millisecond at the 99th percentile, stopping at throughputs that gave results above this threshold.

1. Technical Set-Up - Prerequisites

- You should be able to set-up and test Aeron Transport as described above
- Some provisioned VMs or machines - get in touch with us if you would like to use our Aeron benchmark infrastructure provisioning modules to help with setting this up.

2. Technical Set-up Instructions

Running Aeron Cluster benchmarks requires at least 4 virtual machines to be provisioned. Follow the instructions in the OS setup section on how to configure the operating system for those machines.

Assuming that you already have the benchmarks repository cloned (see Benchmark scripts section on the instructions on how to do that) you can proceed with creating the following wrapper script on the local machine in the benchmarks project (i.e. in the `aeron-benchmarks/scripts` directory).

Let's create a file named `test-cluster.sh`.

```
export SSH_CLIENT_USER=<SSH user>
export SSH_CLIENT_KEY_FILE=<private SSH key file>
export SSH_SERVER_NODE=
export SSH_SERVER_USER=${SSH_CLIENT_USER}
export SSH_SERVER_KEY_FILE=${SSH_CLIENT_KEY_FILE}
export SSH_CLIENT_NODE=<SSH IP of the client node>
export SSH_CLUSTER_NODE0=<SSH IP of the node 0>
export SSH_CLUSTER_NODE1=<SSH IP of the node 1>
export SSH_CLUSTER_NODE2=<SSH IP of the node 2>
export SSH_CLUSTER_USER0=${SSH_CLIENT_USER}
export SSH_CLUSTER_USER1=${SSH_CLIENT_USER}
export SSH_CLUSTER_USER2=${SSH_CLIENT_USER}
export SSH_CLUSTER_KEY_FILE0=${SSH_CLIENT_KEY_FILE}
export SSH_CLUSTER_KEY_FILE1=${SSH_CLIENT_KEY_FILE}
export SSH_CLUSTER_KEY_FILE2=${SSH_CLIENT_KEY_FILE}
CLIENT_NODE_IP=<CLIENT_NODE_IP>
NODE0_IP=<Cluster node 0 IP>
NODE1_IP=<Cluster node 1 IP>
NODE2_IP=<Cluster node 2 IP>


JAVA_HOME=<JAVA_HOME on all nodes>
BENCHMARKS_PATH=<benchmarks path on all nodes>
DATA_DIR=<data directory for storing the Cluster log>
```

```
_CPU_NODE=<CPU node (socket) to run the processes on>
_NON_ISOLATED_CPU_CORES=<non-isolated CPU cores>
_DRIVER_CONDUCTOR_CPU_CORE=<CPU core to pin the ‘conductor’ thread>
_DRIVER_SENDER_CPU_CORE=<CPU core to pin the ‘sender’ thread>
_DRIVER_RECEIVER_CPU_CORE=<CPU core to pin the ‘receiver’ thread>
_ARCHIVE_RECORDER_CPU_CORE=<CPU core to pin the ‘archive-recorder’ thread>
_ARCHIVE_REPLAYER_CPU_CORE=<CPU core to pin the ‘archive-replayer’ thread>
_ARCHIVE_CONDUCTOR_CPU_CORE=<CPU core to pin the ‘archive-conductor’ thread>
_CONSENSUS_MODULE_CPU_CORE=<CPU core to pin the ‘consensus-module’ thread>
_CLUSTERED_SERVICE_CPU_CORE=<CPU core to pin the ‘echo-service’ thread>


export CLUSTER_ID=42
export CLUSTER_SIZE=3
export CLUSTER_BACKUP_NODES=0


export CLIENT_JAVA_HOME=”${JAVA_HOME}”
export CLIENT_BENCHMARKS_PATH=”${BENCHMARKS_PATH}”
export CLIENT_DRIVER_CONDUCTOR_CPU_CORE=${_DRIVER_CONDUCTOR_CPU_CORE}
export CLIENT_DRIVER_SENDER_CPU_CORE=${_DRIVER_SENDER_CPU_CORE}
export CLIENT_DRIVER_RECEIVER_CPU_CORE=${_DRIVER_RECEIVER_CPU_CORE}
export CLIENT_LOAD_TEST_RIG_MAIN_CPU_CORE=${_ARCHIVE_RECORDER_CPU_CORE}
export CLIENT_CPU_NODE=${_CPU_NODE}
export CLIENT_NON_ISOLATED_CPU_CORES=${_NON_ISOLATED_CPU_CORES}
export CLIENT_AERON_DPDK_GATEWAY_IPV4_ADDRESS=
export CLIENT_AERON_DPDK_LOCAL_IPV4_ADDRESS=
export CLIENT_EGRESS_CHANNEL=”aeron:udp?endpoint=${CLIENT_NODE_IP}:0”
export CLIENT_INGRESS_CHANNEL=”aeron:udp”


CONSENSUS_CHANNEL=”aeron:udp?term-length=64k”
_client_ingress_endpoints=””
_cluster_consensus_endpoints=””
_cluster_members=””
for (( n=0; n<CLUSTER_SIZE; n++ ))
do
 ip_var=”NODE${n}_IP”
 if [[ “$n” -ne 0 ]]
 then
   _client_ingress_endpoints+=”,”
   _cluster_consensus_endpoints+=”,”
   _cluster_members+=”|”
 fi
 _client_ingress_endpoints+=”${n}=${!ip_var}:2${n}000”
 _cluster_consensus_endpoints+=”${!ip_var}:2${n}001”
 _cluster_members+=”${n},${!ip_var}:2${n}000,${!ip_var}:2${n}001,${!ip_
var}:2${n}002,${!ip_var}:2${n}003,${!ip_var}:2${n}004”


 export “NODE${n}_JAVA_HOME=${JAVA_HOME}”
 export “NODE${n}_BENCHMARKS_PATH=${BENCHMARKS_PATH}”
 export “NODE${n}_CLUSTER_DIR=${DATA_DIR}/cluster”
 export “NODE${n}_ARCHIVE_DIR=${DATA_DIR}/archive”
 export “NODE${n}_DRIVER_CONDUCTOR_CPU_CORE=${_DRIVER_CONDUCTOR_CPU_CORE}”
 export “NODE${n}_DRIVER_SENDER_CPU_CORE=${_DRIVER_SENDER_CPU_CORE}”
 export “NODE${n}_DRIVER_RECEIVER_CPU_CORE=${_DRIVER_RECEIVER_CPU_CORE}”
 export “NODE${n}_ARCHIVE_RECORDER_CPU_CORE=${_ARCHIVE_RECORDER_CPU_CORE}”
 export “NODE${n}_ARCHIVE_REPLAYER_CPU_CORE=${_ARCHIVE_REPLAYER_CPU_CORE}”
 export “NODE${n}_ARCHIVE_CONDUCTOR_CPU_CORE=${_ARCHIVE_CONDUCTOR_CPU_CORE}”
 export “NODE${n}_CONSENSUS_MODULE_CPU_CORE=${_CONSENSUS_MODULE_CPU_CORE}”
 export “NODE${n}_CLUSTERED_SERVICE_CPU_CORE=${_CLUSTERED_SERVICE_CPU_CORE}”
 export “NODE${n}_CPU_NODE=${_CPU_NODE}”
 export “NODE${n}_NON_ISOLATED_CPU_CORES=${_NON_ISOLATED_CPU_CORES}”
```

```
 export “NODE${n}_AERON_DPDK_GATEWAY_IPV4_ADDRESS=”
 export “NODE${n}_AERON_DPDK_LOCAL_IPV4_ADDRESS=”
 export “NODE${n}_CLUSTER_CONSENSUS_CHANNEL=${CONSENSUS_CHANNEL}”
 export “NODE${n}_CLUSTER_INGRESS_CHANNEL=aeron:udp”
 export “NODE${n}_CLUSTER_LOG_CHANNEL=aeron:udp?term-length=64m|control-
mode=manual|control=${!ip_var}:2${n}002”
 export “NODE${n}_CLUSTER_REPLICATION_CHANNEL=aeron:udp?endpoint=${!ip_
var}:2${n}022”
 export “NODE${n}_ARCHIVE_CONTROL_CHANNEL=aeron:udp?endpoint=${!ip_var}:2${n}004”
 export “NODE${n}_ARCHIVE_REPLICATION_CHANNEL=aeron:udp?endpoint=${!ip_
var}:2${n}044”
done
export CLIENT_INGRESS_ENDPOINTS=”${_client_ingress_endpoints}”
export CLUSTER_CONSENSUS_ENDPOINTS=”${_cluster_consensus_endpoints}”
export CLUSTER_MEMBERS=”${_cluster_members}”


“${DIR}/aeron/remote-cluster-benchmarks” --client-drivers “java,c” --server-drivers
“java,c” --file-sync-level 0 --mtu 1408 --context “my-cluster-test”
```

Now to run the script from with the `aeron-benchmarks/scripts` directory do:

```
$ ./test-cluster.sh
```

ii. Test Results - Summary

The results of the test were as follows:

- For latency, we measured a round trip time of 235 microseconds when using Aeron open source. With Aeron Premium, we saw that latency almost halved - to 130 microseconds at the 99th percentile for 100,000 messages per second of a 288-byte message.
- For throughput, we maintained over 250,000 288-byte messages a second with Aeron open source while staying under our one millisecond threshold. This compares with over 2,000,000 messages a second with Aeron Premium. This is an eight-fold improvement over already incredibly impressive results.

iii. Test Results - Detailed

The tables and charts below give a more detailed view of the results we achieved testing Aeron Cluster using AWS cluster placement groups. If you have questions, please get in touch.

Table: Aeron Cluster Max Throughput (99th percentile less than 1ms on c5n.9xlarge)

	32 bytes	288 bytes	1344 bytes
Java	500K	250K	200K
C	350K	250K	180K
C with DPDK (Aeron Premium)	6M	2M	450K
Aeron Premium ratio compared with C	17.1	8	2.5

Figure: Aeron Cluster Max Throughput for a 32-byte message with 99th percentile less than 1 millisecond (throughput denoted in driver label in the key)

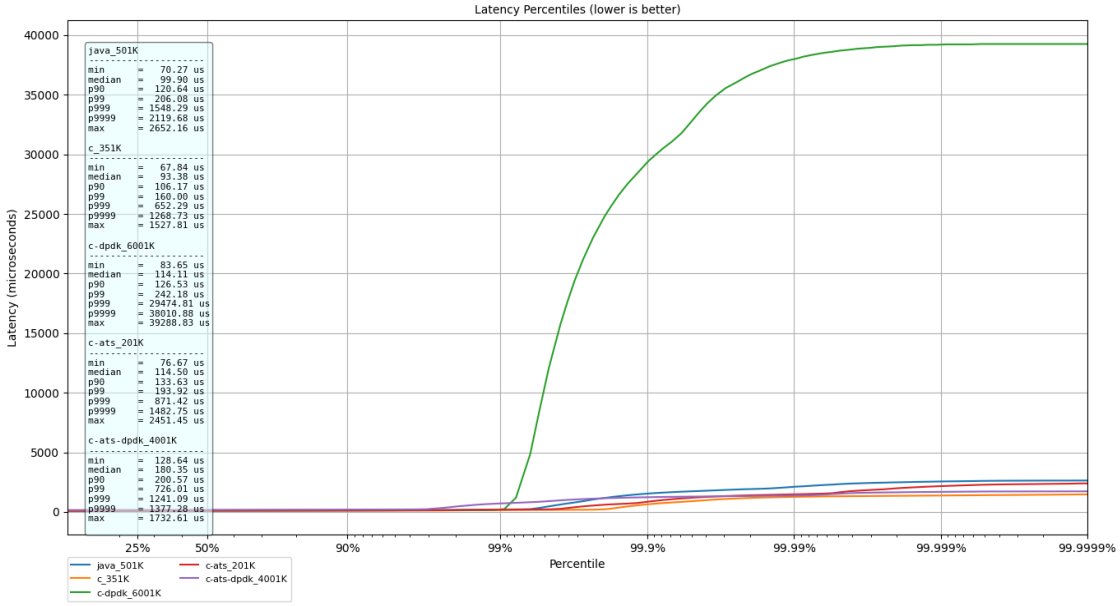


Figure: Aeron Cluster Max Throughput for a 288-byte message with 99th percentile less than 1 millisecond (throughput denoted in driver label in the key)

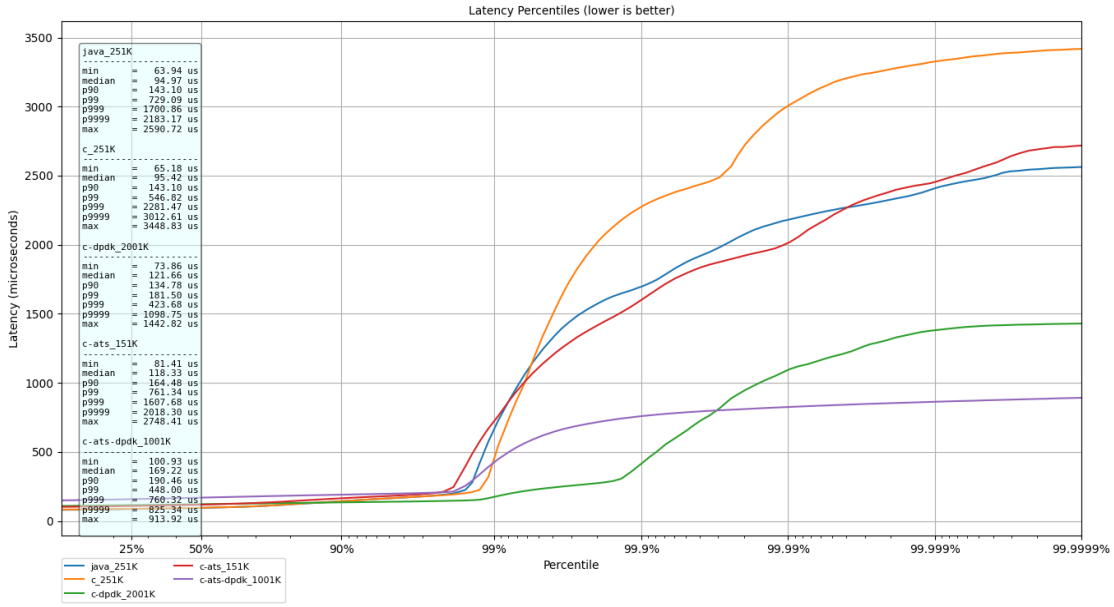


Figure: Aeron Cluster Max Throughput for a 1344-byte message with 99th percentile less than 1 millisecond (throughput denoted in driver label in the key)

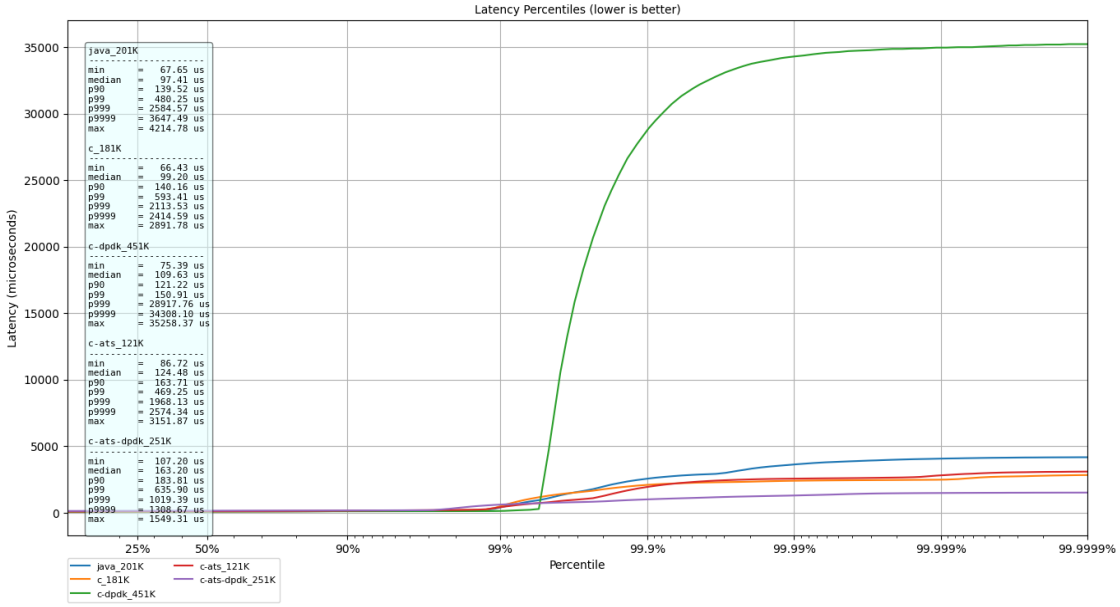


Table: Round trip latency @ 100,000 32 byte msg/sec (µs)

	P50	P99	P999	Max
Java	107	132	150	4657
C	106	130	145	4180
C-DPDK	103	124	137	1823
Aeron Premium ratio compared with C	0.972	0.954	0.945	0.444

Figure: Aeron Cluster round trip latency for a 32-byte message at 100,000 messages per second

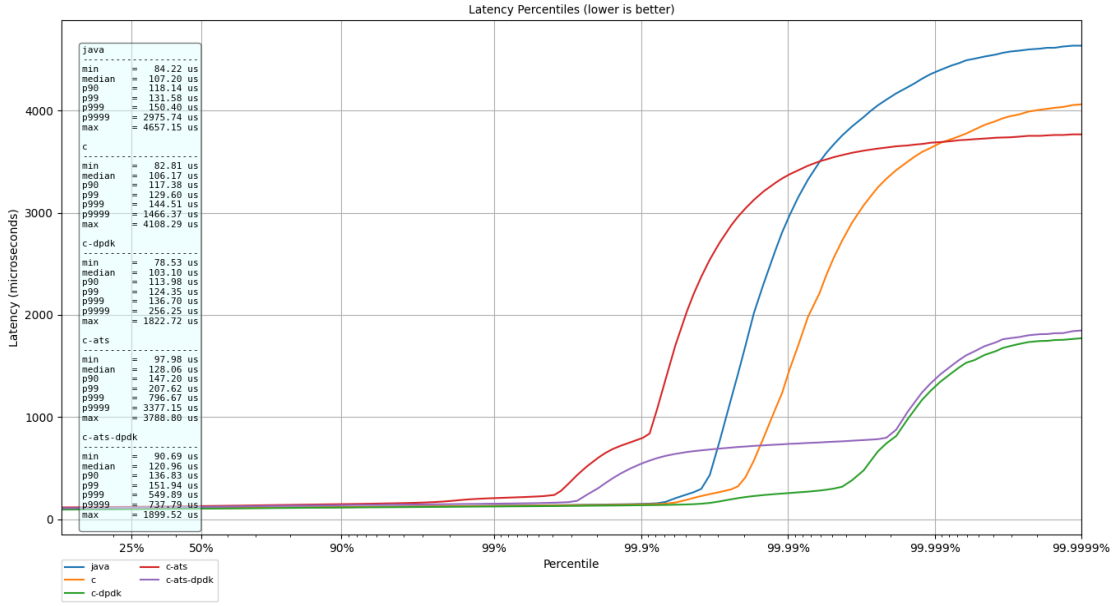


Table: Round trip latency @ 100,000 288byte msg/sec (µs)

	P50	P99	P999	Max
Java	109	235	2408	53706
C	108	239	35520	89194
C-DPDK	106	130	144	1137
Aeron Premium ratio compared with C	0.981	0.544	0.004	0.013

Figure: Aeron Cluster round trip latency for a 288-byte message at 100,000 messages per second

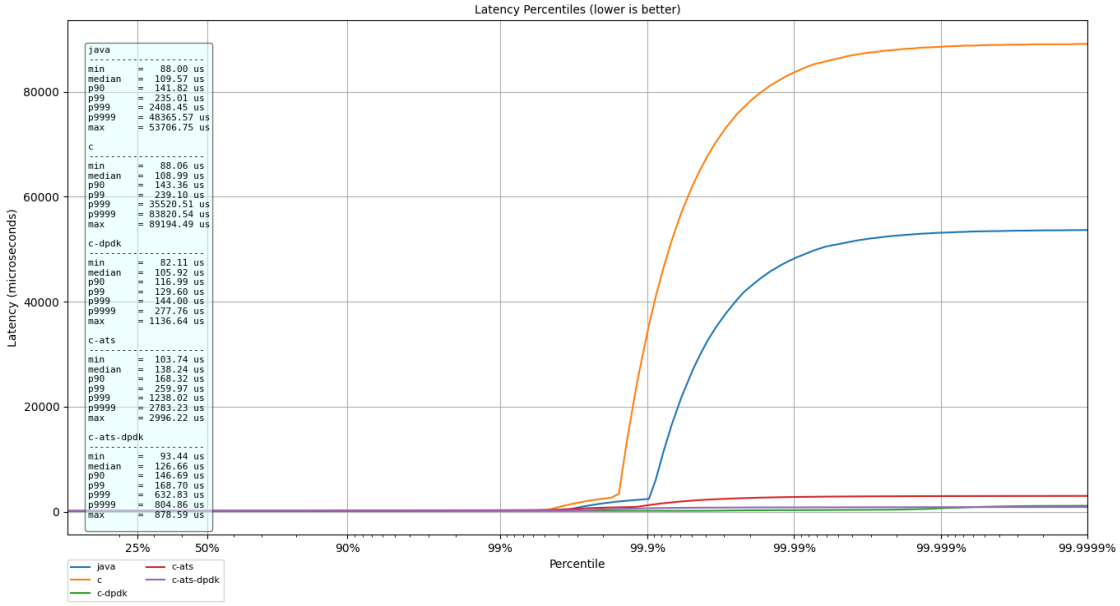
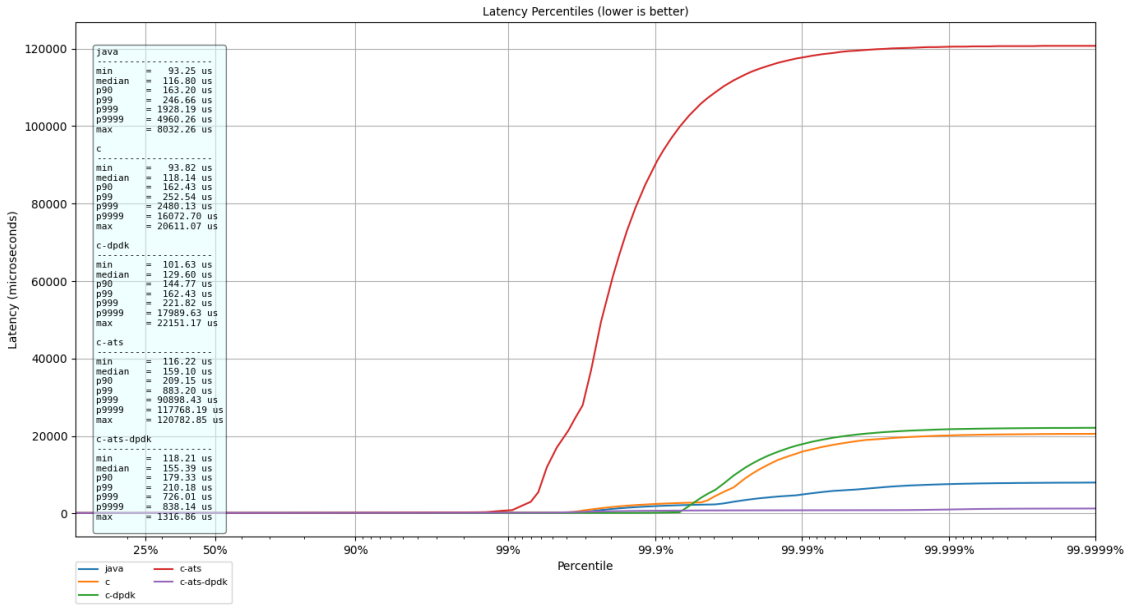


Table: Round trip latency @ 100,000 1344 byte msg/sec (µs)

	P50	P99	P999	Max
Java	117	247	1928	8032
C	118	253	2480	20611
C-DPDK	130	162	222	22151
Aeron Premium ratio compared with C	1.102	0.640	0.090	1.075

Figure: Aeron Cluster round trip latency for a 1344-byte message at 100,000 messages per second



05 Testing Resources

If you need support setting up and running a test or a Proof of Concept (PoC) using Aeron we'd be happy to help. Please get in touch with us to discuss your requirements.

Useful Links and resources:

1. Aeron Cookbook Quick Start Guide >>

2. Videos & Presentations:

    a. Fault Tolerant 24/7 Operations with Aeron Cluster >>

    b. Aeron: Open-source high-performance messaging >>

    c. 24/7 State Replication >>

Appendix

Further Aeron Cluster Testing

The test set-up described in this appendix was also run as a part of the Aeron performance testing on AWS.

i. Test Set-Up 2 : Partition Placement Group - a set-up optimized for redundancy

We deployed Aeron Cluster nodes across different Availability Zones within the same region. That means that messages sent to the cluster are replicated to a quorum of other nodes across at least two other AZs.

This setup gives enhanced reliability with reduced RTO and RPO times, but comes with a trade-off in performance. The latency of transmitting data to nodes in other Availability Zones is simply higher than within a single Availability Zone (more information on the AWS AZ construct can be found here).

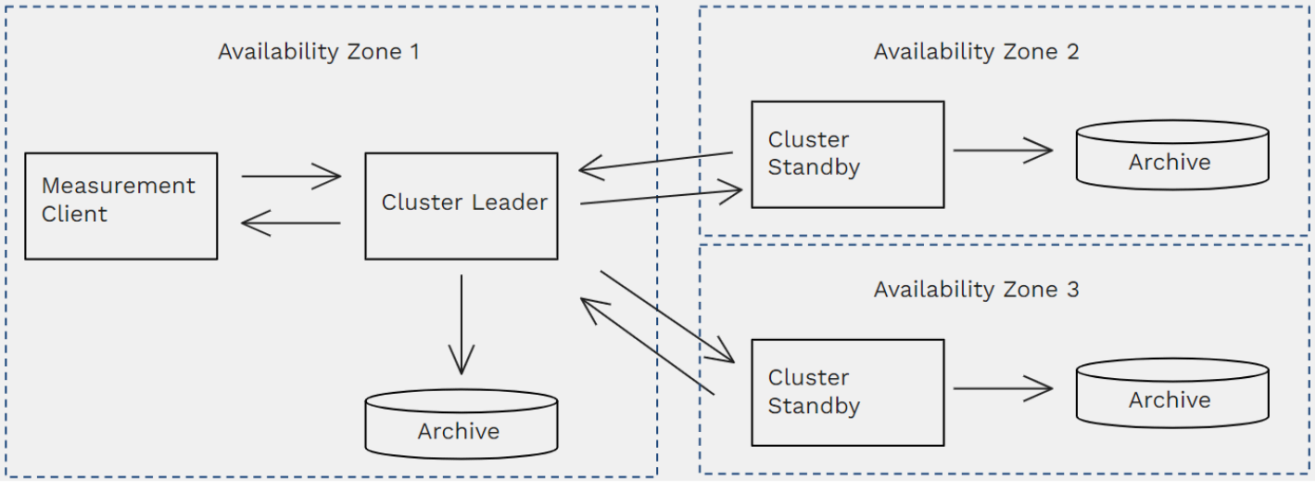


Figure: Aeron Cluster Test Set-up using AWS Partition Placement Group Set-up

For our throughput tests, we increased the acceptable latency threshold before we disregarded our test results. This was simply to account for the increased network latency from having the cluster deployed across Availability Zones. The latency threshold was set to 10 milliseconds, at the 99th percentile.

ii. Test Results - Summary

- For latency, we measure a round trip of 3,428 microseconds using Aeron open source, at the 99.9th percentile. With Aeron Premium the latency almost halved, to 2,109 microseconds, again at the 99.9th percentile, for 100,000 messages per second of a 288-byte message.
- For throughput, with Aeron open source we sustained 250,000 288-byte messages per second. Using Aeron Premium, throughput improved nearly 7x, to over 1,700,000 messages per second.

iii. Test Results - Detailed

The tables and charts below give a more detailed view of the results we achieved testing Aeron Cluster using AWS partition placement groups. If you have questions regarding these, please get in touch.

Table: Aeron Cluster Max Throughput (99th percentile less than 10ms on c5n.9xlarge)

	32 bytes	288 bytes	1344 bytes
Java	500K	250K	250K
C	350K	250K	200K
C with DPDK (Aeron Premium)	5M	1.7M	400K
Aeron Premium ratio compared with C	14.3	6.8	2.0

Figure: Aeron Cluster Max Throughput using partition placement group for a 32-byte message with 99th percentile less than 10 milliseconds (throughput denoted in driver label in the key)

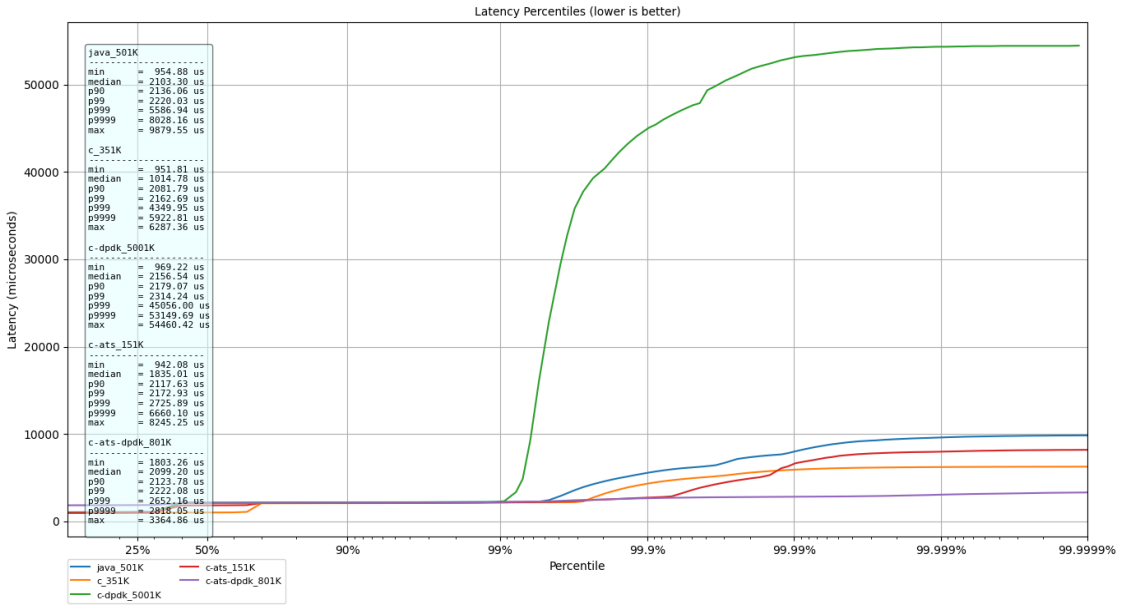


Figure: Aeron Cluster Max Throughput using partition placement group for a 288-byte message with 99th percentile less than 10 milliseconds (throughput denoted in driver label in the key)

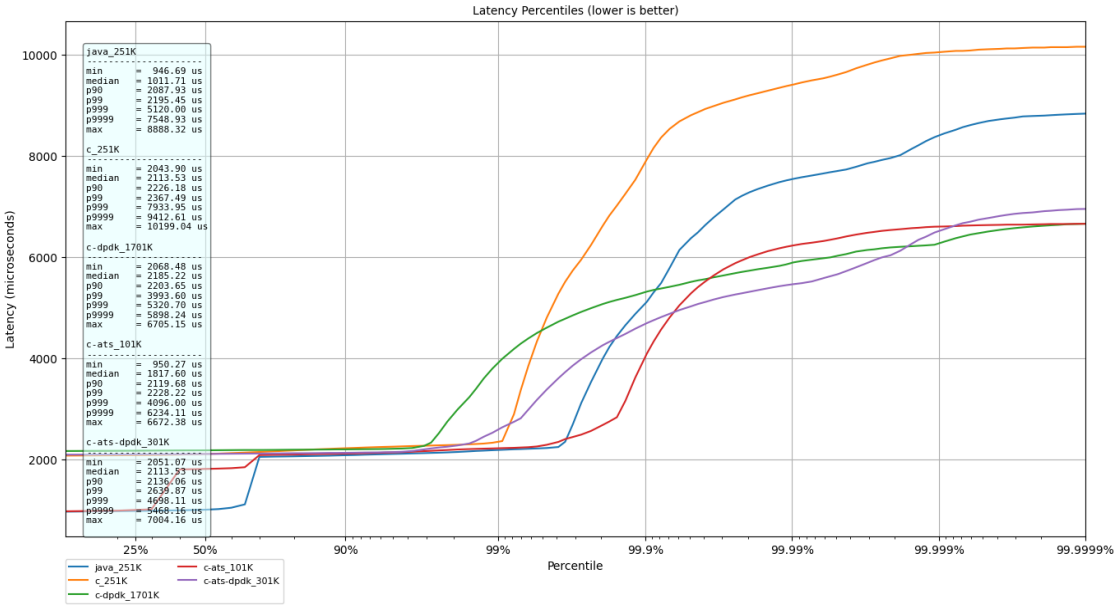


Figure: Aeron Cluster Max Throughput using partition placement group for a 1344-byte message with 99th percentile less than 10 milliseconds (throughput denoted in driver label in the key)

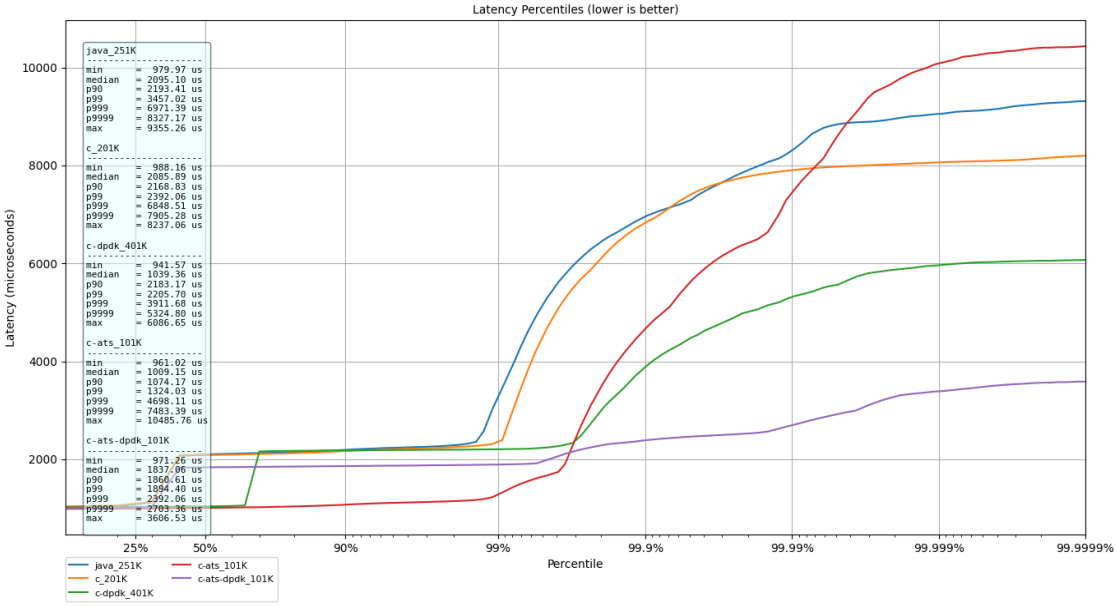


Table: Round trip Latency @ 100k 32-byte msg/sec (µs)

	P50	P99	P999	Max
Java	996	2064	2082	5771
C	2054	2082	2101	5984
C-DPDK	2073	2092	2103	2400
Aeron Premium ratio compared with C	1.009	1.005	1.001	0.401

Figure: Aeron Cluster round trip latency using partition placement group for a 32-byte message at 100,000 messages per second

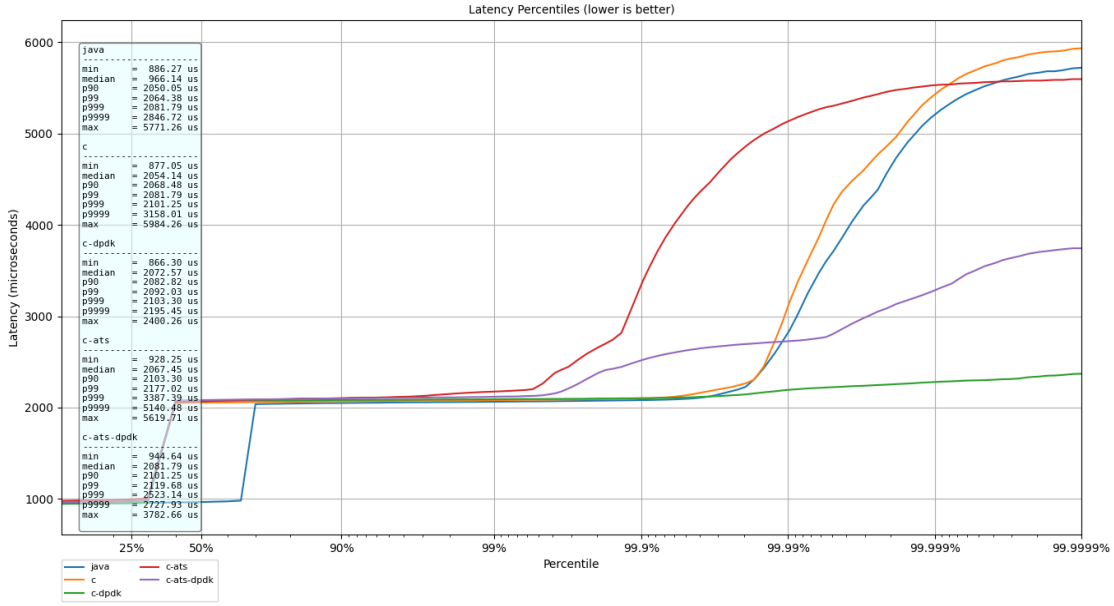


Table: Round trip Latency @ 100k 288byte msg/sec (µs)

	P50	P99	P999	Max
Java	2070	2207	3428	4489
C	2062	2195	3678	7495
C-DPDK	2074	2094	2109	2633
Aeron Premium ratio compared with C	1.006	0.954	0.573	0.351

Figure: Aeron Cluster round trip latency using partition placement group for a 288-byte message at 100,000 messages per second

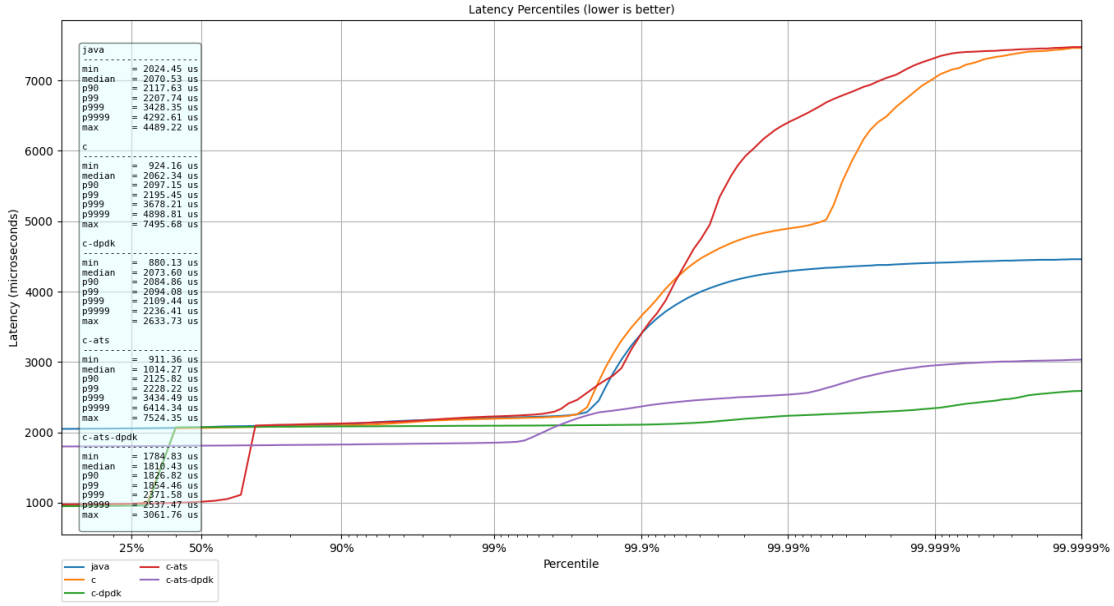
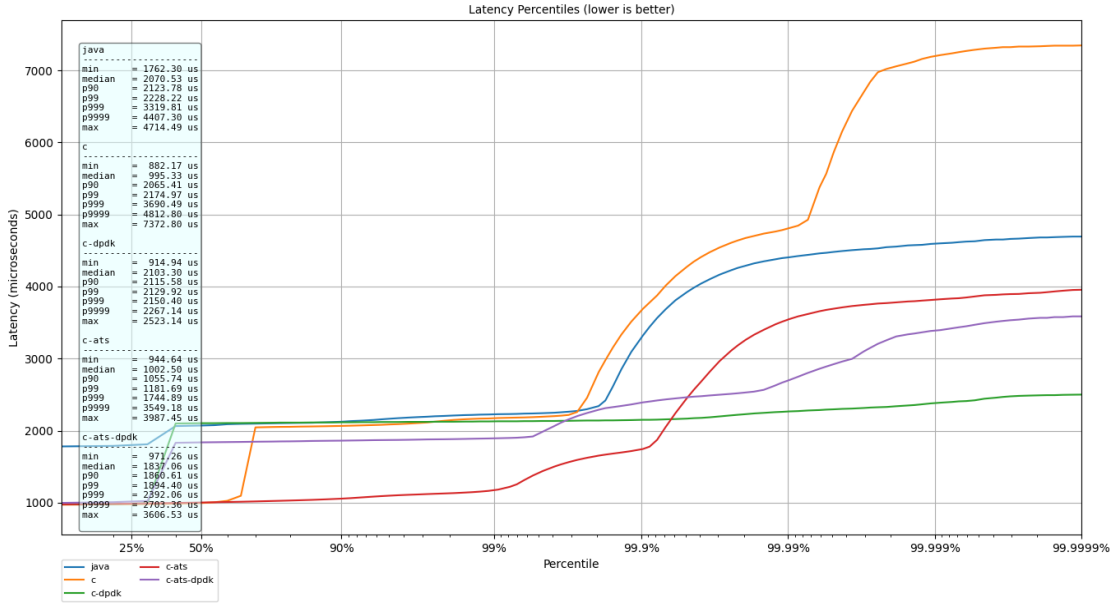


Table: Round trip Latency @ 100k 1344 byte msg/sec (µs)

	P50	P99	P999	Max
Java	2071	2228	3320	4714
C	995	2175	3690	7373
C-DPDK	2103	2130	2150	2523
Aeron Premium ratio compared with C	2.114	0.979	0.583	0.342

Figure: Aeron Cluster round trip latency using partition placement group for a 1344-byte message at 100,000 messages per second



Adaptive builds & operates bespoke trading technology solutions across asset classes for financial services firms wanting to own their technology stack to differentiate and compete in the long-term. Central to Adaptive's offering is Aeron, the global standard for high-throughput, low-latency and fault-tolerant trading systems - the open source technology supported and sponsored by Adaptive.

Weareadaptive.com Aeron.io

Follow us on: