Performance improvements in rosbag2

Adam Dąbrowski (adam.dabrowski@robotec.ai)

Piotr Jaroszek (piotr.jaroszek@robotec.ai)

This report has been contracted by and received funding from Apex.AI, Open Robotics, ADLINK and University of Tokyo.

ROBOTEC.AI
MOKOTOWSKA 1, WARSAW (PL)
E OFFICE@ROBOTEC.AI U WWW.ROBOTEC.AI

Focus of the project

The project was focused on implementing performance improvements to rosbag2 as well as development of a new version of benchmarking package, allowing all users to run highly configurable performance tests.

The most important optimizations that we implemented include asynchronous writing (with size-limited double-buffering) and storage back-end optimization. Some defaults for rosbag2 were also adjusted for better performance.

As a part of the work, we also increased rosbag2 stability, eliminating a couple of sources of crashes due to race conditions. The current rosbag2 did not crash even once for us during extensive benchmarks (while the Foxy version did several times).

Using the developed benchmarking package, we compared rosbag2 after improvements with the former version, which is rosbag2 used in Foxy. We also benchmarked against rosbag1 and included some insights into performance based on different parametrizations.

Benchmarking results

Benchmarking platforms

Two different machines were used for testing. This is a limited sample of devices and a good complementary would be to also benchmark NVMe drives as well as some representative embedded environments.

Benchmarking platform #1

CPU	Intel i7-8750H
Memory	16GB DDR4
Disk	WDS480G2G0B-00EPW0 (SSD Western Digital 480 GB SATA III)
Seq. read	545 MB/s
Seq. write	380 MB/s

Benchmarking platform #2

CPU	Intel i74970k
Memory	16GB DDR3
Disk	CT1000MX500SSD1 (crucial MX500 1TB)
Seq. read	560 MB/s
Seq. write	510 MB/s
Random read IOPS	95 000
Random write IOPS	90 000

In comparison, platform #1 has a better CPU and faster memory, but substantially lower disk write speed. Disk write speed is an important limiting factor in performance of rosbag2.

Important notes

There are some important considerations to keep in mind when interpreting results of benchmarking tests.

1. Disk write speed limit is theoretical and in practice can be affected by many factors.
2. The system on which benchmarks were run was not a clean, “laboratory” environment. OS has its own noise in terms of I/O that can produce disturbances. However, using such system is more akin to what users would experience. We controlled this in following ways:
    a. Browsers and other obvious disk-using application were closed.
    b. Networking was turned off.
    c. We averaged results out of minimum 3 runs.
3. Disks have internal cache which affects short time benchmarks. Results with low total volume (less than ~3 GB) can be particularly strongly affected by this.
    a. We also run more extensive recording benchmarks with up to 50GB of total data recorded

Message counting was done through checking the *metadata.yaml* file. However, we also checked the database size and the benchmark results generated by the package contain the information on file size regardless of whether *preserve_bags* parameter is set to true or false.

Reproducing results

A crucial part of the work is *rosbag2_performance_benchmarking package*. It resides within rosbag2 project and can be built with *--cmake-args -DBUILD_ROSBAG2_BENCHMARKS=1* flag. You can refer to the package *README.md* file for more insights.

Benchmarks can be executed with or without transport layer. In the second case, the data is produced and then put directly into queues for recorder (does not go through publish/subscribe). Benchmarks can be highly customized to a specific data profile (reflecting a use case) and use Yaml parameter files as well as ros2 launch system.

Configuring benchmarks

The rosbag2_performance_benchmarking package enables users to set a profile of publishers for benchmarking. Configurations are set through Yaml config files. The concept of publisher groups is implemented to allow easy testing of multiple similar publishers. The *topic_root* parameter is used to generate topics (with sequence numbers). QoS can also be set (and will be matched by rosbag2 recorder subscriptions).

```yaml
rosbag2_performance_benchmarking_node:
  ros__parameters:
    publishers: # publisher_groups parameter needs to include all the subsequent groups
      publisher_groups: [ "10Mbs_many_frequent_small", "100Mbs_large" ]
      wait_for_subscriptions: True
      10Mbs_many_frequent_small:
        publishers_count:   500
        topic_root:         "benchmarking_small"
        msg_size_bytes:     100
        msg_count_each:     2000
        rate_hz:            200
      100Mbs_large:
        publishers_count:   1
        topic_root:         "benchmarking_large"
        msg_size_bytes:     10000000
        msg_count_each:     100
        rate_hz:            10
        qos:  # qos settings are ignored for writer only benchmarking
          qos_depth:          5
          qos_reliability:    "best_effort" # "reliable"
          qos_durability:     "volatile" # "transient_local"
```

Aside from configuration of publishers / data producers, benchmarks are controlled by a benchmarking configuration Yaml, for example:

```yaml
rosbag2_performance_benchmarking:
  benchmark_node:
    ros__parameters:
      benchmark:
        summary_result_file:  "results.csv"
        db_root_folder:       "rosbag2_performance_test_results"
        repeat_each:          2     # How many times to run each configurations (to average results)
        no_transport:         True  # Whether to run storage-only or end-to-end (including transport) benchmark
        preserve_bags:        False # Whether to leave bag files after experiment (and between runs). Some configurations c
        parameters:                 # Each combination of parameters in this section will be benchmarked
          max_cache_size:         [10000000, 100000000]
          max_bag_size:           [0]
          compression:            ["", "zstd"]
          compression_queue_size: [1]
          compression_threads:    [0]
          storage_config_file:    ["", "storage_resilient.yaml"]
```

You can set the number of experiments to run for each setting (to determine an average as well as statistical variations), whether to preserve bags (but note extensive benchmarking can result in a lot of big files filling up your disk space), and where to put the results.

The *benchmark.paramters* section is used to create and benchmark all possible combinations of parameters which are specified as lists. These results are included in a CSV file, which you can process yourself or use a handy *report_gen.py* script included in the package.

Comparing rosbag2 to Foxy version

Considerations

Comparing against Foxy could only be done with the considering and controlling for the following differences:

- Semantics of the *--max-cache-size* parameter changed from Foxy. In Foxy, it was a count of messages to cache. In master, it is a specified size in bytes.
    - We have set the parameter for Foxy tests to match the size of cache used for master branch tests (calculating the number of messages corresponding to a given size).
- Default durability QoS incompatibility - running master publishers with Foxy ros2 bag record causes warnings about incompatible Durability QoS and no messages recorded by default.
    - We have set durability for publishers to transient local as it was necessary to run the tests.
- Lack of by-regex topic filtering - we use by-regex recording to only record our benchmark topics, e. g. omitting /rosout and /parameter_events.
    - Foxy bags were recorded with -a parameter. Logging was disabled and no traffic on the two additional topics (/rosout and /parameter_events) was recorded, so this difference most likely had no substantial effect on the experiment.

Foxy comparison: benchmarked configurations

Benchmarking was done with the following configurations:

Name	#publishers	msg size	rate (Hz)	msg count	total data size
100MBps	20	200KB	25	1000	4 GB
200MBps	20	400KB	25	1000	8 GB
300MBps	20	600KB	25	1000	12 GB

Note that msg count is per topic. Experiments in this section were run on **Benchmarking platform #1** (380 MB/s theoretical write speed limit).

For the first batch of tests, we used 500 MB cache for master and corresponding cache message counts for Foxy bag - 2500, 1250, 833, as well as bag splitting on 1GB of data.

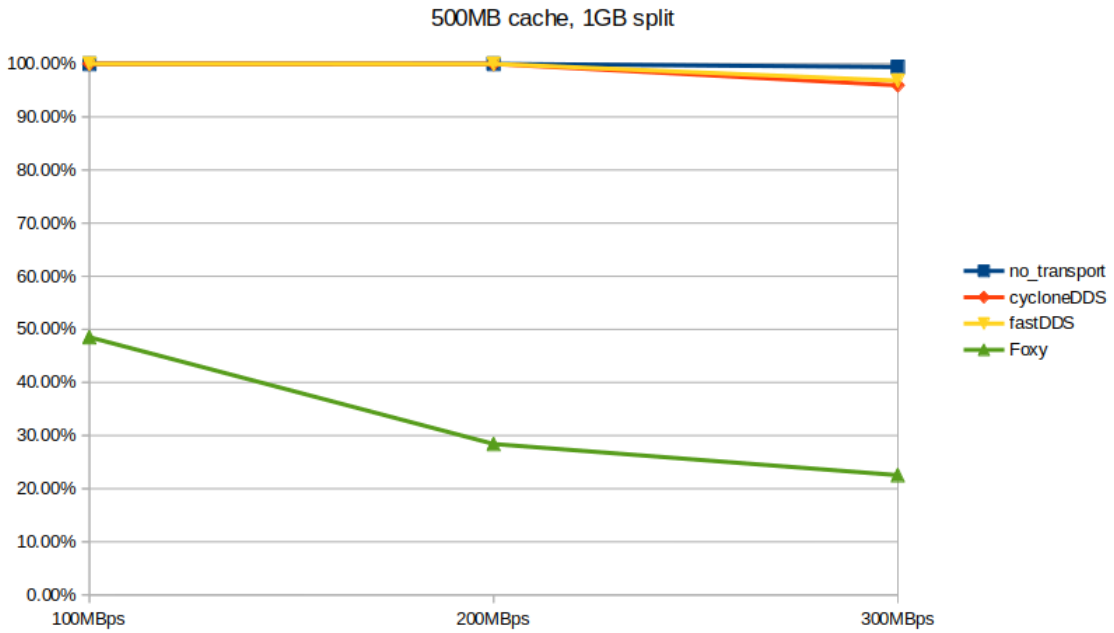
For the second batch, we compared the default settings as these are very important since they are likely going to be left unchanged for many users. This means no splitting and default cache sizes for master (100MB) and Foxy (0 messages).

We ran benchmark publishers and bag separately (to be able to use Foxy bag version). Benchmark command example: *ros2 run rosbag2_performance_benchmarking benchmark_publishers --ros-args --params-file config/producers/300MBs_raw.yaml*

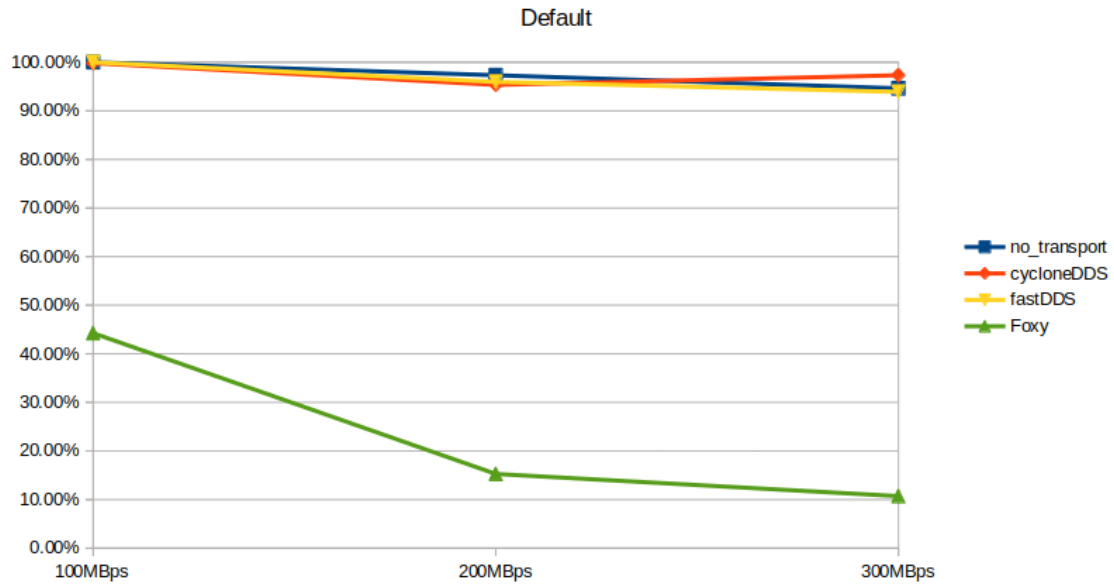
Additionally, **Foxy ros2 bag record crashed twice** during benchmarks (no crashes for master in any of benchmarks) - these were not counted towards averages and repeated. The following error occurred, corresponding to a known and fixed (in master) concurrency issue:

[ERROR] [1612444515.302877599] [rosbag2_transport]: Failed to record: Error when processing SQL statement. SQLite error (11): database disk image is malformed

Foxy comparison: results with record parameters



Foxy comparison: results with defaults for recording



Observations

- Improvements in performance over Foxy are huge
- Raising cache size might be important for larger throughputs

Comparing with rosbag (ROS 1)

It is difficult to meaningfully compare rosbag with rosbag2 since comparison will include transport layer differences and is also dependent on several factors such as:

- Which DDS implementation is used for ros2, and how it is parametrized.
- Intra-process communication optimizations (e. g. shared memory) if running on the same machine.
- Differences in implementations of parameters semantics (e. g. cache, bag splitting).

For the comparison, we locally adapted publishers code from *rosbag2_performance_benchmarking* package to work with ROS(1) rosbag. This is not normally supported by the package and requires some custom work. With such reservations, the following results should be treated with a dose of skepticism, but still serve to capture some measure of overall performance.

Benchmarks were executed on **Benchmarking platform #2**.

Parameters *--buffer* and *--max-cache-size* were set to 500MB for rosbag and rosbag2.

Small scale benchmark

The case of 100 publishers sending 50 KB size messages with 100 Hz frequency was benchmarked (total 500 MB/s, at a hardware theoretical limit). Benchmarks were only for 1000x100 messages (5GB total). Results are averaged from 10 runs:

rosbag2	without transport	100%
	with FastDDS [1]	100%
	with CycloneDDS [1]	97.87%
rosbag	-	100%

[1] Note that performance of ros2 transport can depend a lot on vendor-specific configuration for DDS, e. g. to fit expected message size. With larger throughputs, **it is likely that one will experience higher loss** if DDS configuration is not adapted – e. g. with messages of 500 KB (at 10Hz) score for rosbag2 with CycloneDDS drops to 92.97% with default configuration in small-scale benchmarks. Benchmarking middleware has been done multiple times before and it is outside the scope of this work.

rosbag2 vs rosbag: Large scale stress benchmarks

Note that we ran benchmarks with both FastDDS and CycloneDDS in their default configurations, but the data should not be used to compare DDS implementations, since depending on what data profile (especially the message size for dominant publishers) is used, we repeatedly found that **one or the other turned out to yield more % of recorded messages**.

Benchmark setup

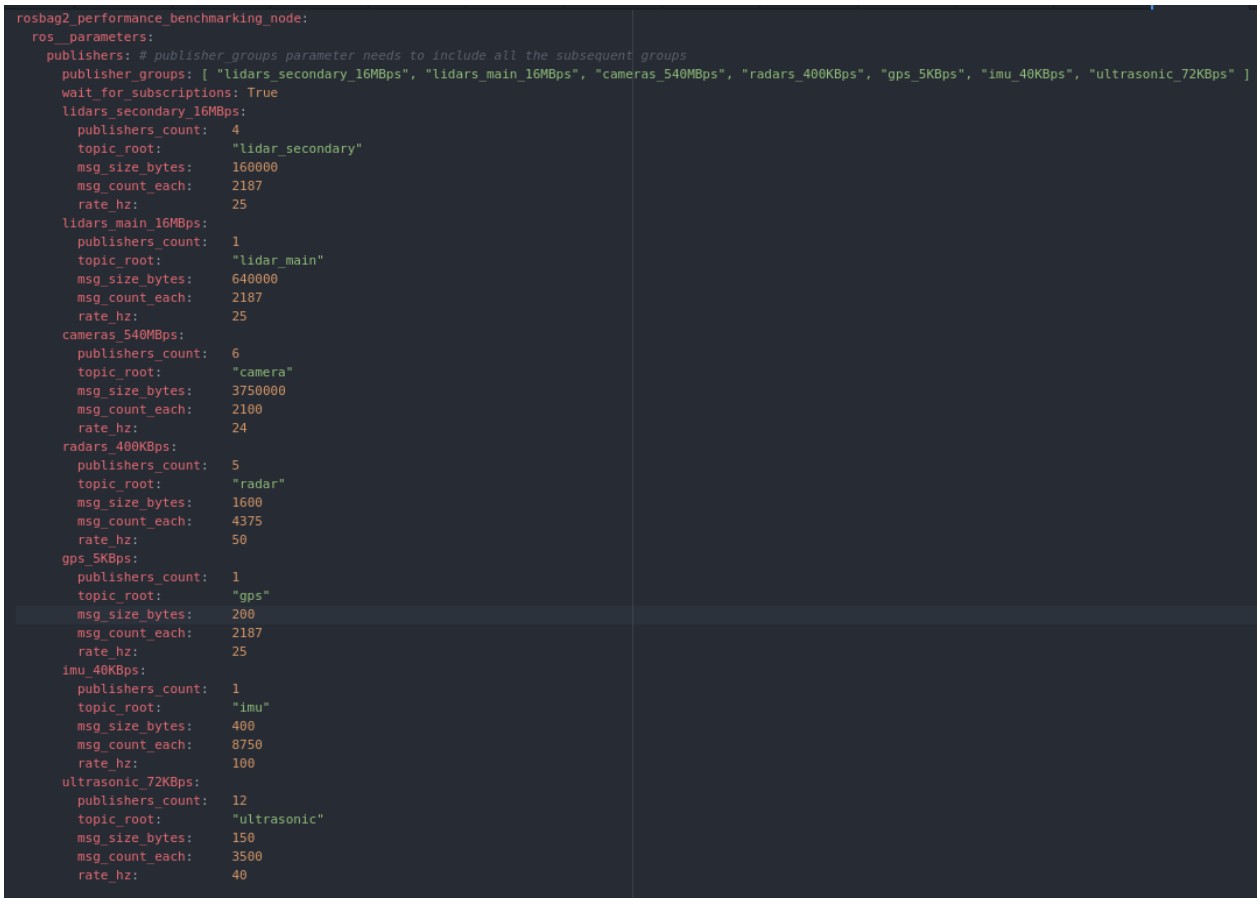
For this benchmark, we increased throughput to and over the disk write speed limits as well as expanded benchmark time to total 50GB of recorded data. This way we almost removed the influence of internal disk cache and tested both stability and performance in extreme cases. We also set higher values for both publisher count and topic rates. A case of 100MBps was also added as a control for this change of parameters. A case called "Automotive" was also benchmarked, containing a mix of topics that could typically be found with AV sensors' output recording.

Name	#pubs	msg size	rate (Hz)	msg count	total data size
100MBps	100	10KB	100	50000	50 GB
350MBps	100	35KB	100	14285	50 GB
500MBps	100	50KB	100	10000	50 GB
Automotive (570 MBps)	30 total	various	various	various	50 GB
600MBps	100	60KB	100	8333	50 GB

5 runs for each configuration were averaged, and the *--max-cache-size* parameter set to 500MB.

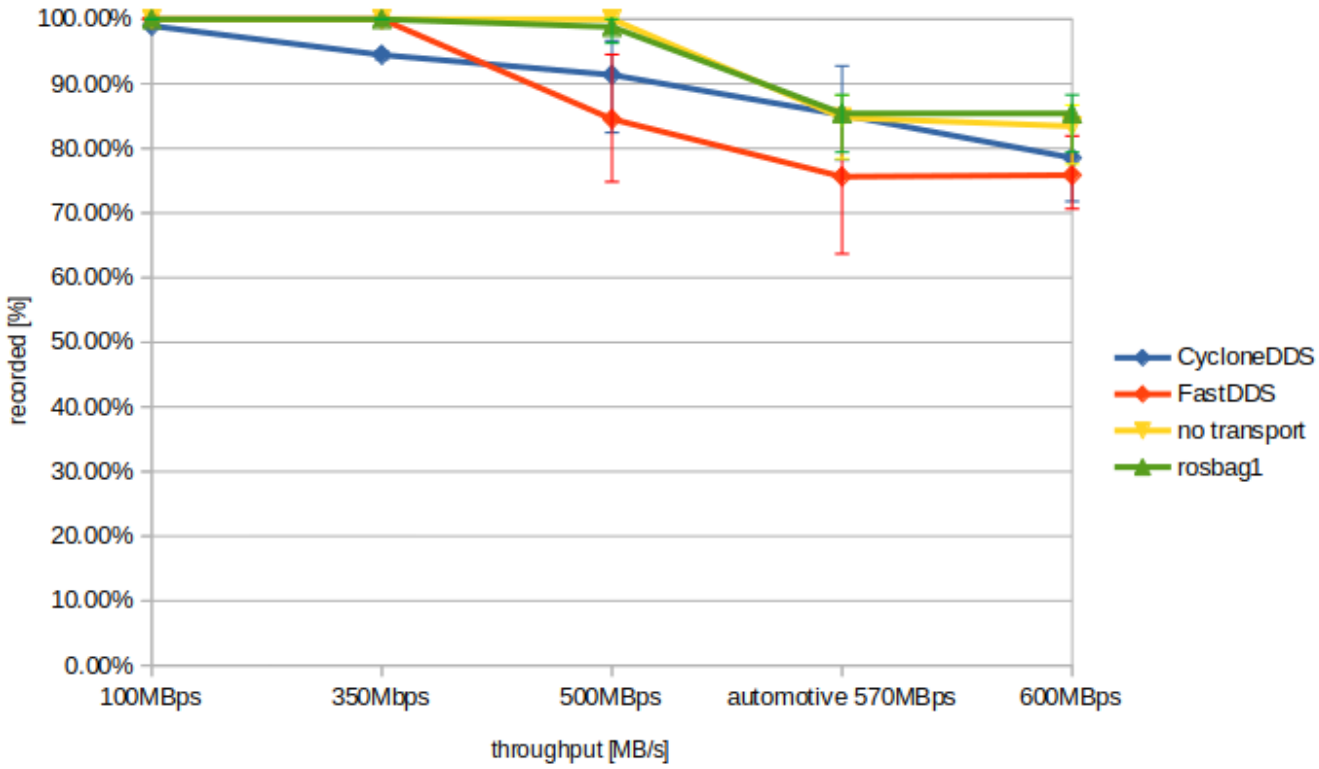
rosbag2 vs rosbag: “Automotive” configuration

A mix of publishers prepared with an intention of being a more natural test case:



```
rosbag2_performance_benchmarking_node:
  ros__parameters:
    publishers: # publisher_groups parameter needs to include all the subsequent groups
      publisher_groups: [ "lidars_secondary_16MBps", "lidars_main_16MBps", "cameras_540MBps", "radars_400KBps", "gps_5KBps", "imu_40KBps", "ultrasonic_72KBps" ]
      wait_for_subscriptions: True
      lidars_secondary_16MBps:
        publishers_count:   4
        topic_root:         "lidar_secondary"
        msg_size_bytes:     160000
        msg_count_each:     2187
        rate_hz:            25
      lidars_main_16MBps:
        publishers_count:   1
        topic_root:         "lidar_main"
        msg_size_bytes:     640000
        msg_count_each:     2187
        rate_hz:            25
      cameras_540MBps:
        publishers_count:   6
        topic_root:         "camera"
        msg_size_bytes:     3750000
        msg_count_each:     2100
        rate_hz:            24
      radars_400KBps:
        publishers_count:   5
        topic_root:         "radar"
        msg_size_bytes:     1600
        msg_count_each:     4375
        rate_hz:            50
      gps_5KBps:
        publishers_count:   1
        topic_root:         "gps"
        msg_size_bytes:     200
        msg_count_each:     2187
        rate_hz:            25
      imu_40KBps:
        publishers_count:   1
        topic_root:         "imu"
        msg_size_bytes:     400
        msg_count_each:     8750
        rate_hz:            100
      ultrasonic_72KBps:
        publishers_count:   12
        topic_root:         "ultrasonic"
        msg_size_bytes:     150
        msg_count_each:     3500
        rate_hz:            40
```

rosbag2 vs rosbag: stress results

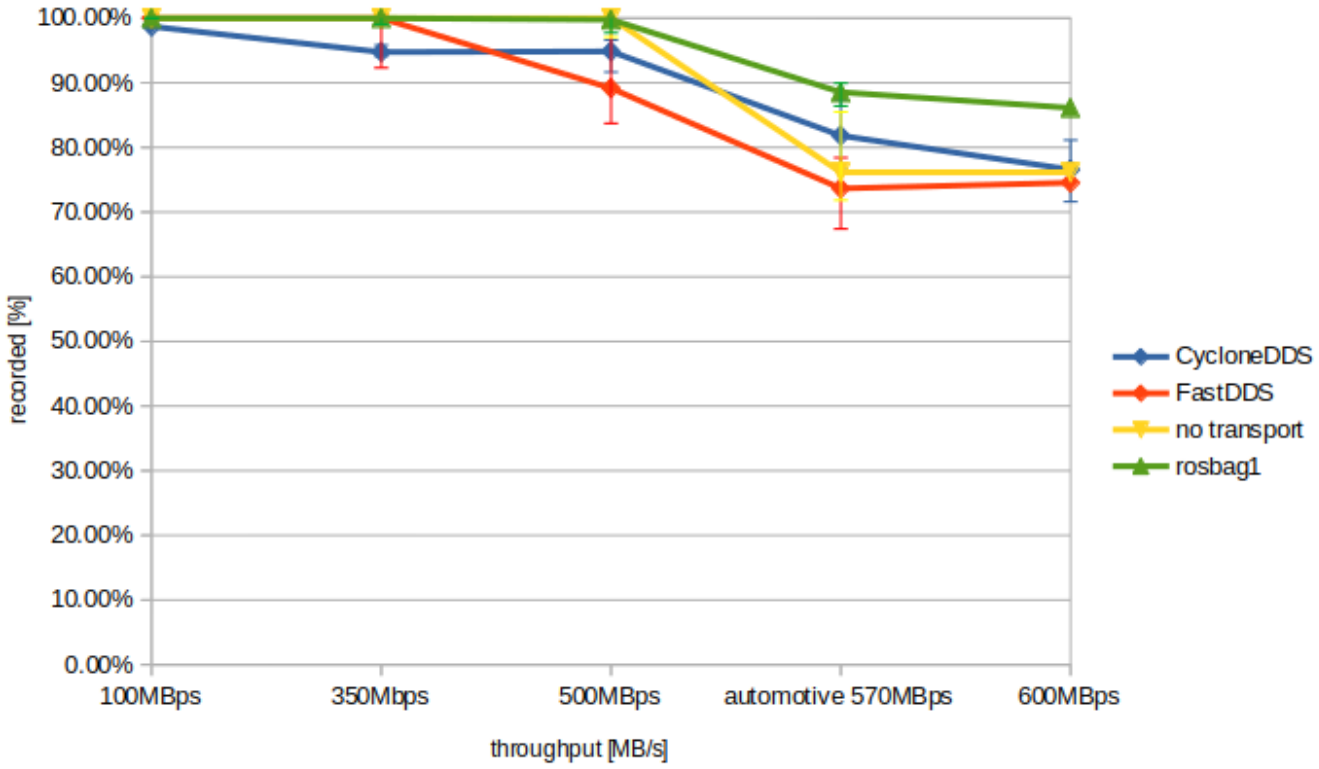


Notes

- All data points besides the first (100MBps) are at or above the disk write speed limit. These are stress tests.
- No-transport case is very close to rosbag1, indicating that most of the loss in extreme cases happens in transport.
- In a more natural, "Automotive" stress use-case, with CycloneDDS, performance is very much the same as rosbag1.
- Transport for both FastDDS and CycloneDDS can be configured to perform optimally in specific use cases. This comparison is for the default.
- Data profile for benchmarks was not selected to favor any DDS implementation.
- Performance is measured by count of messages received, not by the total size of missing data. This is important in the "Automotive" case.

rosbag2 vs rosbag: stress results with splitting

Same setup as before, but with split on 1GB bag size.



Notes

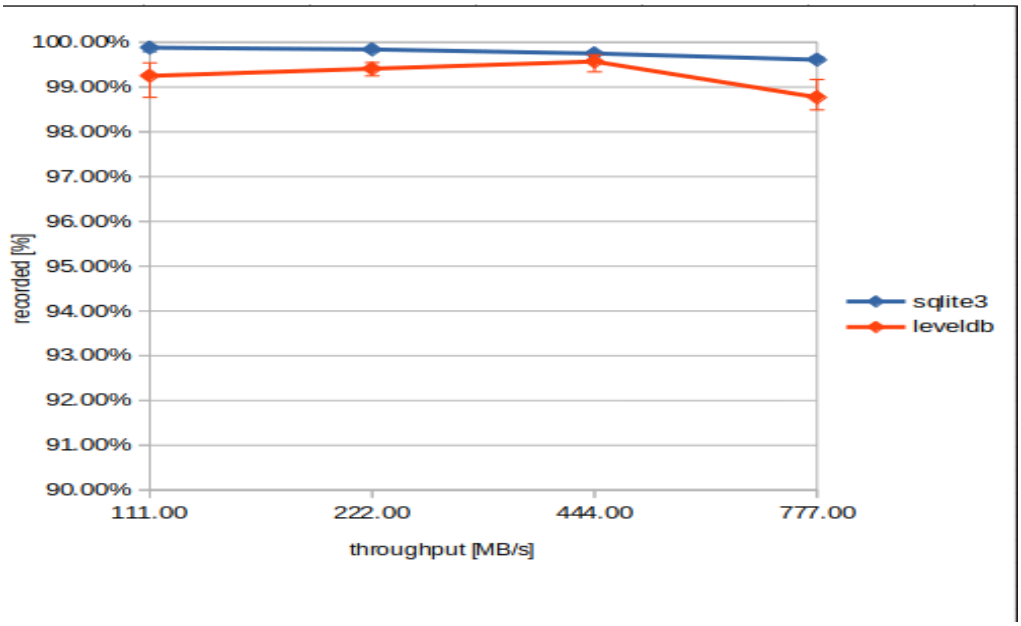
- Bag splitting seems to improve the performance a bit in the 500MBps (at-hardware-limit case).
- Bag splitting seems to lower the relative performance in extreme cases, which are evidently handled a bit better by rosbag1.
- Bag splitting obviously affects the performance on the storage level, as suggested by reduced performance in the no-transport case.
    - Issue #640 contains reasoning on why this is happening.
    - Significant CPU time is used up in should_split_bagfile() in SequentialWriter.
- There is an anomaly with no-transport for the "Automotive" case. It is uncertain why this happened, an average of 5 runs is not diminished by an obvious outlier.

SQLite3 vs leveldb

A limited set of benchmarks in this chapter were conducted on **Benchmarking platform #2** with CycloneDDS (default on the master branch).

Note that the performance is surprisingly good in extreme cases since the internal disk cache is an important factor with less data recorded in total. However, the comparison is not invalidated by the fact, as the intention was to compare the performance of the storage stack. A more comprehensive set of benchmarks should be conducted to reach confidence in conclusions. A mix of large and small messages was benchmarked.

Throughput test: sqlite3 vs leveldb



- Note: **Y axis starts with 90%**. Cache is set to 500 MB. Ten seconds of data is sent (so, 1.1 GB, 2.2 GB, 4.4 GB, 7.7 GB total).
- For big data publishers, qos_reliablity is set to best_effort
- Recording copes quite well even in cases surpassing the I/O limit.
    - Drops mostly big messages. This is likely because of:
        - QoS of best_effort
        - Double-buffer implementation of cache limit is naturally working against big messages when stressed. This can be an issue in some cases and a preferred outcome in other cases.
- Sqlite3 seems to cope slightly better than leveldb in terms of % of message dropped (default configurations were used for both). The difference is small.

Observations and conclusions

- Performance of rosbag2 was drastically improved from Foxy to master
- Performance of rosbag2 seems to be on par or almost on par with rosbag (ROS1), depending on the data profile.
- Overall, rosbag2 performs quite well with high throughputs across diverse configurations.
    - Losses in scenarios with throughputs lower than disk write speed are relatively small.
    - Losses are biased towards best-effort (as expected) and larger (as a side effect of cache) messages in stressful cases.
- Setting cache parameter to about 1 second of total data going through seems to be a good practice.
- Rosbag2 does not crash anymore (after we fixed a couple of race conditions) even when very stressed.
- The discovery & subscription part of recording takes a considerable time (order of second(s) with hundreds of publishers). The effect could be that the messages sent before subscription are lost. This can be partially offset by specifying topics (as opposed to capturing –all or with –regex) and completely offset by waiting for subscription using Node count_subscribers().
    - We made waiting for subscribers an optional parameter in benchmarks.
    - We run our benchmarks with this flag on since it eliminates a known and by-design message loss mechanism.
- A small fraction of messages is sometimes lost in with-transport tests even in medium-throughput scenarios. Example:
    - 1 publisher, 1Mb, 100Hz, sending 1000 messages, cache 500MB (100MB/s)
    - Out of 20 attempts, 4 had 999/1000 messages captured, the rest was 100%.
- Transport loss is highly dependent on configuration and can be greatly reduced (increasing receive buffer size, datagram max size, adjusting queue size in QoS)
- Losses occurring outside of transport are registered (even per topic). Losses occurring in transport (including rmw implementations) are not currently counted or logged in any way.

Suggested work

1. Introduce a CI job for performance, using benchmarking packages on a dedicated (at least exclusive at the time of the test) system.
2. Run benchmarks on NVMe drives as well as on embedded platforms.
3. Run a deeper investigation of transport-related losses and count losses on writer and reader queues and overwrite events separately.
4. Run very large experiments (Terabytes of data)
5. Benchmark recording with compression. The packages support this, but currently messages are random data (not compressible) so tests won't be representative unless data is changed to realistic.
6. Improve performance of bag splitting as suggested by issue #604.
7. Rething some defaults in configuration for middleware as well as the cache size parameter default – it is currently 100MB, ros1 has a 256MB buffer, and high throughput cases are much better off with 500MB or so.

Acknowledgements

This report has been contracted by and received funding from Apex.AI, Open Robotics, ADLINK and University of Tokyo.

ROBOTEC.AI
MOKOTOWSKA 1, WARSAW (PL)
E OFFICE@ROBOTEC.AI **U** WWW.ROBOTEC.AI

Appendix I: Effects of specific optimizations

We have checked the impact of each optimization group separately. This was done with smaller samples on **Benchmarking platform #2.**

Effects of double buffering optimization

A custom double buffering with a memory-size-limited queues was introduced to implement asynchronous storage writing. The effect on performance was measured through benchmarks, which were executed 3 times each with various setups of topics count (instances), message size and *--max-cache-size* parameter.

This compares **only** the effect of double buffering, without back-end optimizations yet:

Results of rosbag2_performance_writer_benchmarking tests						
Disk write (hardware) limit: 500 MB/s. Each percentage in cell is an average of 3 tries.						
100 MB/s of data cases					Performance:	
Sending 2000 messages per instance (publisher). Parameters of test:					Recorded messages %	
instances	freq	msg size [KB]	cache [MB]		Master	This PR
1000	100	1	1		58.66%	68.71%
100	100	10	1		93.85%	76.99%
10	100	100	1		86.63%	75.55%
1	100	1000	1		90.17%	69.67%
1000	100	1	100		50.55%	90.84%
100	100	10	100		65.55%	100.00%
10	100	100	100		67.20%	100.00%
1	100	1000	100		69.03%	100.00%
1000	100	1	500		50.55%	100.00%
100	100	10	500		67.98%	100.00%
10	100	100	500		70.37%	100.00%
1	100	1000	500		71.07%	100.00%
500 MB/s of data cases					Performance:	
Sending 2000 messages per instance (publisher). Parameters of test:					Recorded messages %	
instances	freq	msg size [KB]	cache [MB]		Master	This PR
1000	100	5	1		24.45%	17.47%
100	100	50	1		25.08%	17.46%
10	100	500	1		29.49%	16.89%
1	100	5000	1		22.47%	21.28%
1000	100	5	100		25.55%	29.10%
100	100	50	100		29.17%	33.88%
10	100	500	100		29.55%	34.22%
1	100	5000	100		30.55%	33.07%
1000	100	5	500		25.55%	36.32%
100	100	50	500		26.85%	37.88%
10	100	500	500		27.46%	39.99%
1	100	5000	500		29.95%	38.05%

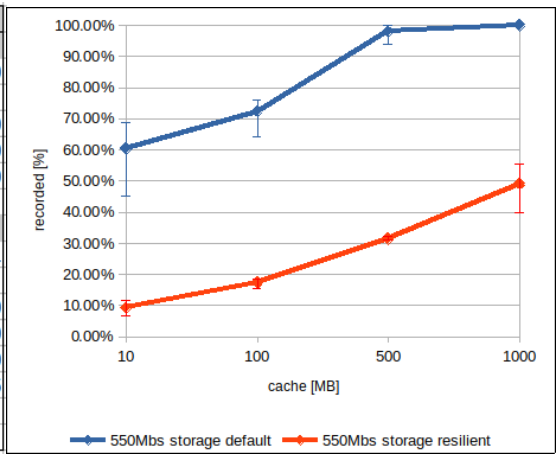
Note how before the optimization, even in 100 MB/s case rosbag2 was not able to record all messages (by a significant margin). Also, increasing cache parameter was not previously helpful and even worsened the situation since the messages were lost in series (each cache flush was blocking subscribers' callbacks processing) instead of more evenly. 1 MB/s is a pathological parameter value with the new implementation and is included in comparison to show how important it is not to set this parameter too low - 1MB/s is 0.01 seconds and 0.002 seconds of data correspondingly in measured cases. Recommended value is equivalent of 1 second of data. In the process of implementing optimizations, the default for the *--max-cache-size* parameter was changed to 100 MB.

More context: https://github.com/ros2/rosbag2/pull/546.

Effects of optimization of back-end

Back end (sqlite3) was optimized for a better performance. Optimization is now on by default and the *--storage-config-preset* parameter was set to resilient for comparison (this setting restores old behavior). Benchmarking results are with double buffering already in place (for both data series). This test includes transport and benchmarks edge case of 550 MB/s (slightly above hardware limitations) with the following configuration:

550 MB/s			
	50MBs_many_frequent_small:		
		publishers_count	500
		topic_root	benchmarking_small
		msg_size_bytes	500
		msg_count_each	2000
		rate_hz	200
	500MBs_large		
		publishers_count	1
		topic_root	benchmarking_large
		msg_size_bytes	50000000
		msg_count_each	100
		rate_hz	10
		qos_depth	5
		qos_reliability	best_effort
		qos_durability	volatile



Note how the *--max-cache-size* parameter should be high enough (with a good ballpark of a value corresponding to 1 second of data) for optimizations to really kick in. Both 10MB and 100 MB can be considered pathological values for the 550 MB/s case. The most important increases from ~31% to ~98% and from ~49% to almost 100% show how impactful the optimization is.

Together with double buffering, optimizations result in a drastic increase in performance as measured by % of successfully recorded messages: **from ~25%** (note that this shows 550MB/s vs 500MB/s in the previous table) messages recorded **to ~100%.**

Appendix II: rosbag2 submitted PRs

1. https://github.com/ros2/rosbag2/pull/594 - packages that allow to benchmark rosbag2 recording with varying setups and parameters.
2. https://github.com/ros2/rosbag2/pull/568 - optimization of sqlite3 back-end for writing, introduced as a configuration option by our previous PRs, now is a default.
3. https://github.com/ros2/rosbag2/pull/497, https://github.com/ros2/rosbag2/pull/493 - *(Contributions to PRs opened by Karsten)* both for reading optimization pragmas from config file for sqlite3. This opened a big improvement in performance, subsequently made a default by #568.
4. https://github.com/ros2/rosbag2/pull/545 - improvements to handling of config files for storage
5. https://github.com/ros2/rosbag2/pull/546 - custom double buffering for storage writing, significantly improving performance by eliminating waiting through producer-consumer scheme on limited-size queues (controlled by --max-cache-size parameter). Also changing default for this parameter to a better one performance-wise.
6. https://github.com/ros2/rosbag2/pull/603 - a very important fix resolving concurrency issues in storage, found through high-performance cases of our benchmarks and code analysis.
7. https://github.com/ros2/rosbag2/pull/604 - record topics by regex. This supports benchmarking, allowing to record hundreds of topics (but when we don't want to record all, such as rosout or from other ros2 nodes running).
8. https://github.com/ros2/rosbag2/pull/634 - improvements in the performance package.

Appendix III: ros2 bag play

Analyzing the code and running some tests we determined the following.

In general, data that can be recorded can also be played on the same system. Three possible causes of message loss are:

1. Messages which are played before there is a subscription (this is also the case of ROS 1)
2. Messages are lost in transport for various reasons (not specific to rosbag2). With reliable transport medium, QoS and proper implementation of receiving node (correct queue sizes and processing so that there is no loss due to queues being full) this can be minimized.
3. Messages lost due to async publishing (DataWriter) - issue https://github.com/ros2/rosbag2/issues/571. This issue does not affect CycloneDDS (which is using synchronous publish) and is already handled by other people.

Player employs a queue that is filled asynchronously, which has a similar effect to the double buffering that we implemented in the writer. The size is 1000 by default and can be changed, which is worth considering if use case is characterized by a high number of messages per second (large number of publishers and/or high frequencies).

Since the limiting factor for a successful use of a recorded data is still the recording part, **we focused our efforts on optimizing storage writing as well as end-to-end process of recording**. We are far from claiming that rosbag2 play cannot be further upgraded in terms of performance.

Appendix IV: implementation of the work plan

A shared document is used to track meetings and progress: https://docs.google.com/document/d/1pDSK-ySgFd_bVb9_4LguBgqYp9ox3F7RCqNMlH07Mj4/edit?ts=5f7b610a#heading=h.yjk4yyft3cuq.

Achieving project goals

Fixing storage issues identified in the previous report:

1. Enabling SQLite configuration (https://github.com/ros2/rosbag2/issues/437) - Done
    a. Additionally, we set optimization configuration as default and introduced storage presets.
2. Implement asynchronous storage write model (https://github.com/ros2/rosbag2/issues/436) - Done
    a. This has been done in storage implementation agnostic way, so it is reusable for other implementations. Significantly improved performance.
    b. Additionally, we introduced counting of dropped messages per topic.
3. Improve –max-cache-size parameter logic and set a better default (https://github.com/ros2/rosbag2/issues/424) - Done
    a. We decided it is best to cache based on used memory and not the elapsed time
    b. Handling of parameter as size was implemented by someone else before we had a chance to take it, but we implemented the size-limited logic with the double buffering which replaces the former implementation.

Profile and fix transport related issues (for Cyclone DDS):

1. Found and fixed a DDS-independent issue with discovery, which resulted in crashes due to race condition in unprotected non-thread safe stl collections. This was found thanks to benchmarking with large number of topics (issue: https://github.com/ros2/rosbag2/issues/602, fix: https://github.com/ros2/rosbag2/pull/603) - note that this resolves two independent concurrency issues. - Done
2. Informed Cyclone DDS team about issue with using more than 118 publishing threads. This hard limit will be changed in a further Cyclone release, but it is not an important issue for a typical user - Done

3. We are in the process of investigating where exactly the transport loss in high traffic benchmarks occurs - Done (thanks to Eric).

Profile and fix the end-to-end system:

1. Test small size data types sent with high frequency mixed with large data – Done
    a. Additionally, we enabled benchmarking with fully configurable publisher setups (to replicate a given use-case).
2. Test with QoS settings – Done
    a. Additionally, we enabled setting of QoS for all publishers in the benchmarking package
3. Test recording of parameters – Done
4. Not directly mentioned in the initial plan but relevant to complete profiling:
    a. Benchmarks validating recording with compression and support of compression settings in our benchmarks – Done.
        - Note that since we are using random data, we observed that compression worsened performance a bit and did not check how it copes with typical data.
    b. Benchmarks validating bag splitting – Done.
    c. Comprehensive benchmarks of effects of various parameters on message loss – Part done (not that comprehensive)
    d. We did not provide support for rosbag2 play benchmarking and only run a few simple tests and performed code analysis – Only put small effort.
        - Analysis and rationale behind are presented in a separate chapter in this report.

Additional items for if there is time left:

1. Vary the number of producer nodes – Not done
2. Record the date from this topology that mimics an application – Done
    a. We implemented a yaml analogue of the json under the link to configure publishers and benchmarks.

Work that was initially planned but discarded in the process:

1. Investigating / fixing issues observed with Autoware.Auto running Dashing - Cancelled
    a. We agreed to not tackle this – rationale is documented in the Tracking Document (under notes from 30 November 2020).

Additional (initially unplanned and/or not formally contracted) work:

1. We implemented filtering recorded topics by regex (--regex and –exclude options) - Done
    a. This was not directly contracted and Robotec.ai initiative. We needed a way to filter out /parameters and /rosout topics in with-transport benchmarks, as well as to ensure that user running another ros2 node does not pollute results. We decided it is best to do it properly.
    b. As it was not directly contracted, can be counted or not as you will, we are fine with treating is as a free contribution (8h of work).
2. Refactored benchmarking packages to use launch files, offer high configurability, and to work smoothly between with-transport and without-transport benchmarking. This was according to the guidance of reviewers. - Done
3. Provided some support with rosbag2 related issues to community and some companies (on request). - Done
4. Benchmarked recording with sqlite3 against leveldb. - Done
5. Benchmarked recoding against rosbag1 - Done