Atlas Fusion 2.0 - A ROS2 Based Real-Time Sensor Fusion Framework

Ing. Stanislav Svědiroh (✉) and prof. Ing. Luděk Žalud, Ph.D.

Department of Control and Instrumentation
Brno University of Technology
CEITEC - Central European Institute of Technology
Brno, Czechia
xsvedi01@vutbr.cz

Abstract. In this paper, we present a novel, easy-to-use ROS2-based real-time sensor fusion framework capable of making high-level detections from raw sensor data provided by their respective drivers. This framework is a direct successor of Atlas Fusion [8] developed by Brno University of Technology robotics lab. As opposed to its predecessor, it is based on ROS2 and more in line with its philosophy - each functionality is encapsulated in its own process (node). This allows for the composition of a unique sensor-fusion pipeline, code testing in isolation, better profiling, and easier usage of the state-of-the-art ROS2 packages developed by other research teams. Algorithms used are real-time, so the framework can be used in development, simulations (with previously collected dataset), deployed to a physical autonomous agent and the high-level detections can be shared between multiple agents. The Atlas-Fusion-2.0 framework has been developed in a way that allows for code distribution between several physical devices which helps with dividing responsibility and building redundancy into the system. With RVIZ and PlotJuggler, one can visualize every part of the processing chain from raw data up to high-level detections to assess current performance. It also has inbuilt basic profiling capabilities to publish the current delay each algorithm introduces to the system. This framework has been evaluated and tested on a sensory framework used to collect the Brno Urban Dataset [6] and its winter extension [7]. As the boundary of the state-of-the-art algorithms in sensor data processing is pushed rapidly, this package, in our opinion, provides a very streamlined way of experimenting with them and testing their performance.

Keywords: ROS2, Real-Time, Sensor Fusion, Framework, Experimentation, Instrumentation, Testing

1 Introduction

In recent years, the field of sensor fusion in robotics has undergone a significant transformation. The rapid advancements in state-of-the-art technology have resulted in sensors capable of capturing and processing high-precision data with

increased information density. However, this enhanced sensing capability comes at the cost of higher computational requirements for extracting appropriate information relevant to specific robotic tasks. While chip manufacturers have made efforts to address this need with faster CPUs and GPUs, there is still a demand for novel approaches to meet the real-time processing requirements of most robotic applications.

Conventional algorithms often fall short of providing a comprehensive understanding of the robot's environment as they struggle to achieve the necessary robustness for practical production use. This limitation necessitates the integration of a wide range of neural network models, which offer the ability to perform advanced analysis and detection on data that was previously inaccessible. By leveraging these neural network models, we can bridge the gap between raw sensor data and high-level interpretations, enabling a more accurate and comprehensive understanding of the robot's surroundings.

Integrating all software packages from sensor drivers and preprocessing algorithms, to fusion and detection algorithms in a clean, testable, expandable and maintainable manner ready to be deployed in a production environment has always been a challenge. The go-to framework in recent years has definitely been ROS2 as it provides the roboticists with tools to split their software solution into separate programs (called "nodes" in ROS2 jargon) and to set up communication between them. This separation of concerns leads, on top of previously mentioned characteristics, to the ability to easily share and reuse code. ROS2, unlike its predecessor ROS1, is fully decentralized, gracefully handles lossy networks and has built-in security mechanisms thanks to the usage of DDS (Data Distribution Service) [12].

ROS2 also supports "Node Composition" - meaning the roboticist can choose, during compile-time or run-time, which nodes should be contained together in a single process and which should be separated. In high-bandwidth paths of the system like image or lidar data, it is beneficial to circumvent the DDS implementation and communicate directly using smart pointers, which results in much greater throughoutput and lower latency which is crucial for every real-time application [10]. Other parts of the system could benefit from separation (either separate processes or even different physical devices on the same network) as it helps with fault isolation and analysis on a smaller scope.

There are several ROS2-based packages that either implement sensor fusion only as a small part of a much larger autonomous navigation system [17,9]. Those packages are very powerful, but their learning curve is steep, and trying to implement and use a novel sensor fusion algorithm will take a lot of time and there is no direct way of benchmarking its performance. On the other hand, there are many packages that implement a very specific algorithm, but their usage on their own does not provide the full picture of the robot's surroundings [14,11]. To take advantage of their popularity, extensive testing, community of contributors and to exploit the inherent property of ROS2, those packages can be integrated into a larger system.

2 Contribution

The goal of this work has been to solve the issue of having to set up a system of ROS2 nodes every time we wanted to test a particular idea or test a new open-source algorithm. We wanted this system to be widely applicable, both for small robots and autonomous vehicles, so we didn't go down the road of previously mentioned bigger frameworks. We are also aware, that if the system needs to have a shallow learning curve and be quick to use, it needs to take advantage of already existing, well-known packages. For this reason, we've created a ROS2 framework consisting of a basic sensor-fusion pipeline incorporating data from GNSS, IMU, Lidars, Radars, RGB and IR cameras that can be easily reconfigured and expanded. At the same time, we've created a very basic tool for code instrumentation, that publishes runtime information straight to ROS2 via lightweight topic. This instrumentation can be turned off when not needed without any changes to the code itself. For deeper tracing analysis we recommend using the ros2_tracing package, which can analyze timings inside the ROS2 middleware layer as well as provide memory usage analysis.

3 Architecture Overview

The architecture of our solution is designed to be as simple as possible, maintaining one way direct path between the raw sensor data on the input and high-level detections on the output. This allows for easy pipeline reorganization, swapping out parts of a pipeline for different algorihms or even running multiple in parallel to compare their performance in the simulation. The block diagram in the figure 1 depicts the overall pipeline. The data frame is first loaded in the driver, ros2 bag or dataloader. More on dataloading in the following section. When needed, it is then preprocessed - in the current version only the lidar data gets preprocessed so the motion distortion gets suppressed [20,13]. Lidar data can also be filtered so the ground plane that doesn't necessarily carry valuable information is stripped off, making following ROS2 messages smaller, thus decreasing communication latency. There could also be a preprocessing step for other sensors, but that's application specific and doesn't apply to our use case. The lidar data are then fused together, possibly normalized to a voxel grid.

Next layer in the architecture is responsible for making the high-level detection out of the preprocessed data. We're currently using two neural networks. One for the detection of vehicles, pedestrians and cyclists in pointclouds and one for rgb camera based detections that can distinguish between even more classes. In our case, with the Brno Urban Dataset [6], a single instance of the pointcloud based neural network is run on the aggregated lidar data and single instance of rgb camera based neural network is run for every camera (4 instances).

On the very end of our simple pipeline, there is a detection matching node to fuse the detections that come from different modalities into a single more robust one.

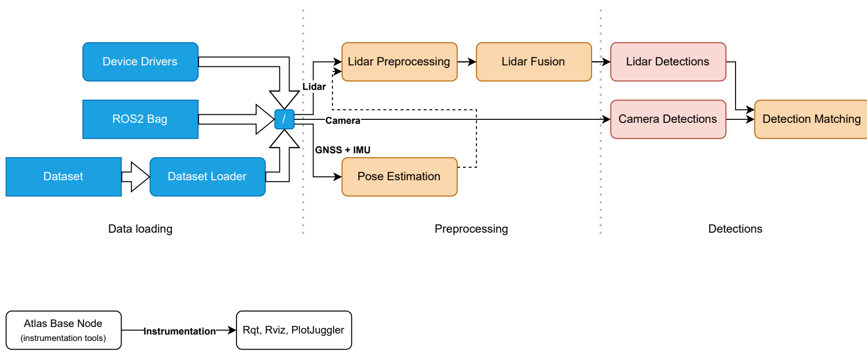


Fig. 1: Atlas Fusion 2.0 - the simplistic extensible pipeline backbone. Nodes built on top of Atlas Base Node (rounded ones) are capable of automatically measuring their runtime performance.

4 Data Loading

Loading the data into the sensor fusion pipeline can be done in various ways depending on the application. As stated above, the minimal pipeline presented is able to run in real-time on physical hardware as well as in the simulation, from previously recorded ROS2 bags or even datasets saved in a different format. Every data frame originates from one of three sources:

(a) Physical or simulated sensor (device driver)
(b) ROS2 bag previously recorded
(c) File on the drive (dataset)

The first option is the most straightforward. Data generated by the sensor are serialized into ROS2 messages and they directly enter the sensor fusion pipeline. This way we can really ensure the end-to-end latencies are suitable for the specific robotic task given the algorithms we are testing. On the other hand, this option isn't very suitable for tuning the algorithms as we cannot easily reproduce the same input data on every run.

For this reason, it is also possible (inherently thanks to the ROS2 framework) to record the data output from the device driver into the ROS2 bag, which is an SQLite database with ROS2 messages in binary format. That way we get easily reproducible, fast to search through, real-world or simulated situations on which the algorithms under test can be evaluated and tuned. One downside is that this can lead to a huge file on your system depending on the number of sensors included in the robotic platform.

Lastly, when using open source datasets, the data usually comes in formats that can be viewed outside of ROS2 (usually .csv for text-based data, .pcd for point clouds and H.264 or H.265 encoded video or individual frames as .jpeg images). For those, it is important that they provide a precise timestamp of the time the data has been taken. In this case, our framework provides an extensible

dataset loader, that can accommodate any practical number of input sensors and outputs data frames into ROS2 in the correct order. The block diagram in the figure 2 shows its working principle.

Every data source (sensor) has its own node, much like the physical sensor would. This node continuously loads data frames from the hard drive to the system memory and reads ahead a pre-defined number of frames. The dataset loader controller is a node that handles the synchronization between all of them, retransmitting the data frames in the correct order and informing them back about the timestamp of the last data frame published. This way only a small amount of data is ever loaded in the system memory. This method, when compared to the previous two, is relatively slow and takes a considerable amount of system resources that could be otherwise utilized by the processing algorithms. For this reason, we often use this only to convert an open-source dataset into the ROS2 bag, which is much quicker to work with.

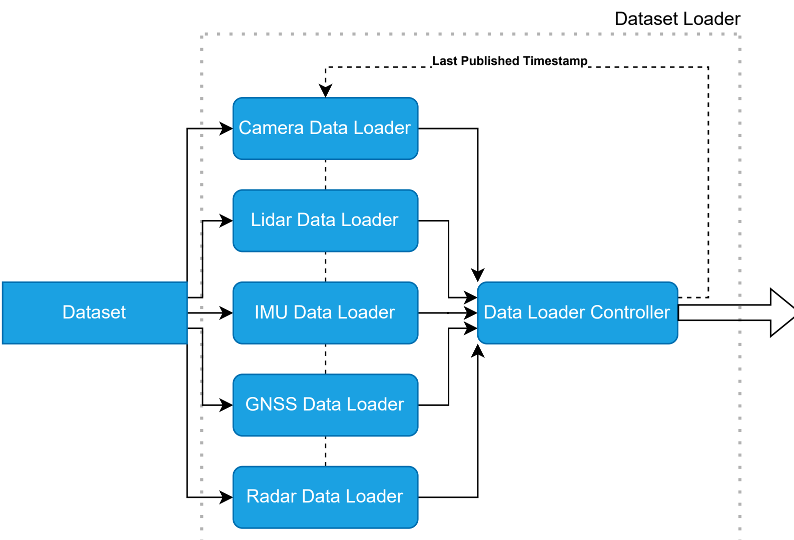


Fig. 2: Block diagram showing how the loading of a proprietary dataset is done enforcing the correct order of data frames read from different sources. This configuration also allows for dynamically adding/removing data sources when needed. This doesn't allow for real-time performance with a practical number of sensors, so recording the output of a Dataset Loader into a ROS2 bag and replaying it with an increased speed is often utilized.

5 Pose Estimation

Precise positioning is a crucial aspect of every robotic task. Without a pose estimate at every point in time, it would be impossible to fuse or aggregate data reliably. Pose estimation is something that has already been researched extensively and software packages that provide a very reasonable performance are readily available. We've used a robot_localization package [14] in this work since it is very popular and has the ability to fuse GNSS with IMU and odometry while having a ton of configuration options. In our case, while using the sensory framework used for the creation of the Brno Urban Dataset, we're fusing an RTK GNSS with IMU.

As we didn't collect any odometry data in this dataset, which would have required hijacking onto the CAN bus in the vehicle, we use the *navsat_transform_-node* from this package to convert our RTK GNSS *sensor_msgs/NavSatFix* messages into the vehicle coordinate frame to get a position relative to its starting point. This node also uses the IMU data to guide its heading estimation. The IMU and the GNSS data transformed into the coordinate frame of the vehicle are then fused using an Extended (EKF) or Unscented (UKF) Kalman filter resulting in a high-frequency pose estimate. Ideally, most robots should have at least one source of odometry, which would make things easier with this package. Robot Localization estimates 15 dimensional state of the robot $(X, Y, Z, roll, pitch, yaw, \dot{X}, \dot{Y}, \dot{Z}, \dot{roll}, \dot{pitch}, \dot{yaw}, \ddot{X}, \ddot{Y}, \ddot{Z})$

There is also an option to generate the odometry from visual or lidar data, but that is not yet implemented in our solution as in most robotic tasks there is a better odometry source that doesn't impose such computational burden. Conceptually, the pose estimation is shown in figure 3.

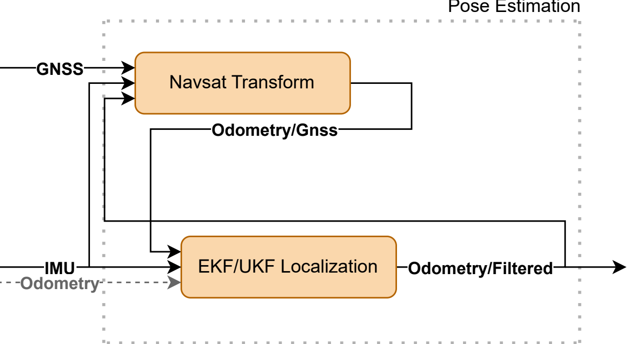


Fig. 3: IMU + GNSS Fusion using the ROS2 *robot_localization* package. To convert GNSS into a local coordinate system the *Navsat Transform* node is used. To get the estimated heading, this node is also aided by the IMU. The final state estimate is done in EKF/UKF localization node. Odometry sources are also supported by this package.

6 Lidar Preprocessing

Lidar data are preprocessed in multiple ways, depending on the application. When using 360° rotating lidar for tasks when the robotic platform movement speeds are not negligible compared to the scanning speed of the lidar, it is necessary to remove the motion distortion, which could significantly affect the following processing algorithms [20,13]. There is a plethora of ways to tackle this problem, but they all have the same foundation - estimating the trajectory between successive lidar scans. As this platform is only experimental, we've settled on assuming that the motion between the lidar scans is linear and since we already have a high-frequency pose estimation output from the *robot_localization* package, we split the incoming point cloud into N batches (each batch represents a sector of a circle) and transform them into the pose estimated at the end of the scan. Figure 4 shows one of the possible defects we need to correct for.

Another optional way of preprocessing point clouds is ground removal. Depending on the task, the ground could carry no relevant information and thus the points only consume valuable bandwidth. For ground robotic platforms, the basic ground removal just filters out points with Z coordinates below the set threshold. From experience, the result of this naive approach is quick and often good enough.

Last, but arguably the most important step is to actually merge point clouds from multiple sensors together. We've, once again, implemented a very basic algorithm because of its fast runtime speeds. A ROS2 node listens to all preprocessed point cloud topics and once it has the latest scan from each of them, it transforms them into a common coordinate system, concatenates and retransmits them. For our use case (2 Velodyne HDL32-e on each side and a forward-pointing Livox Horizon) this effectively creates a relatively dense 360° point cloud with higher density in the most critical direction. Optionally, since for all point cloud manipulations we're using PCL library [16], we can easily downsample the point cloud on a voxel grid using the *pcl::VoxelGrid*.

We've also stripped the core of Spatio-Temporal Voxel Layer (STVL) [11] package from the Nav2 framework [9] it was meant to be used with because of its high customizability and great short-term aggregation capabilities, but since our primary use case during development has been to fuse autonomous vehicle data in real-time, the surrounding environment has been so dynamic, that any short-term aggregation would immediately result in blurring. We've still kept this adapted package for completeness as it can be of great use for different applications.

7 Lidar Detections

The primary reason behind preprocessing the lidar scans has been to allow for an efficient neural-network-based vehicle, cyclist and pedestrian detection. As of the publication of this paper, the quickest at inference time and amongst state-of-the-art in terms of accuracy is the PointPillars encoder [5]. It is the only one that

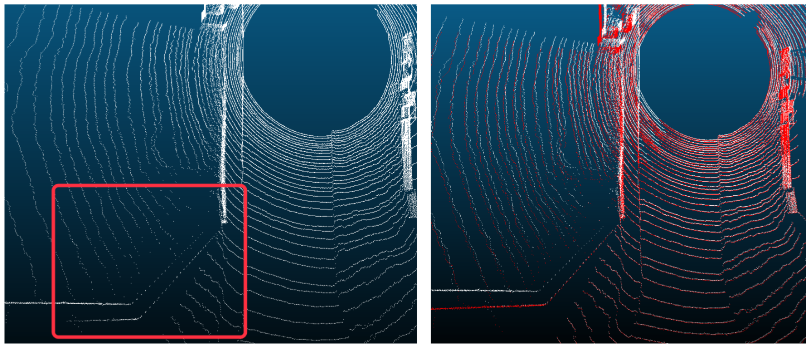
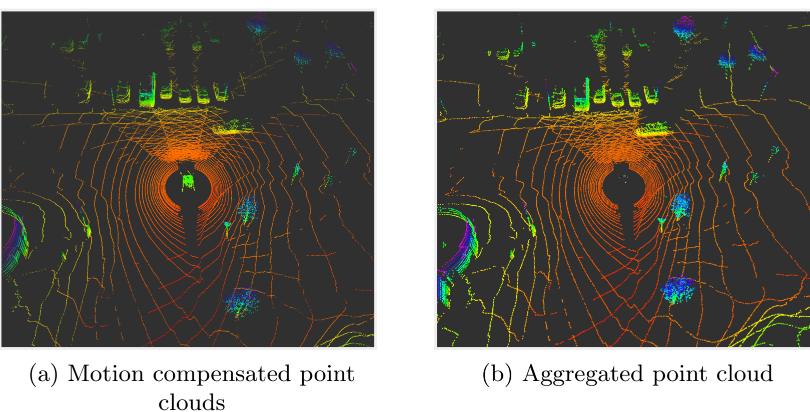


Fig. 4: Merriaux [13] demonstrates one of the possible defects on point clouds while the vehicle is moving. The left image shows a duplicated fence due to the vehicle rotating in the same direction as the lidar. The right image shows the corrected point cloud in red



(a) Motion compensated point clouds

(b) Aggregated point cloud

Fig. 5: The difference between 3 compensated lidar sensor scans and an aggregated downsampled point cloud. Visually there's not much of a difference, but the latter contains about 4x fewer points when downsampled by 15cm voxels

can keep up with the 20Hz lidar scans reliably and even has a 3-6 times margin (60-100 Hz). We've created a Python ROS2 node that utilizes a very helpful framework OpenPCDet [18]. With this implementation of PointPillars and a model trained on the KITTI dataset [3] that it provides, we were able to achieve 10 ms of inference time on an Nvidia RTX 3080 graphics card. OpenPCDet has an abstraction above the point cloud-based neural networks so it is possible and very easy to try out different models, although none of them achieves a real-time performance.

After inference, we're publishing the detections as oriented bounding boxes in the form of *visualization_msgs/MarkerArray* for RViz visualization and *vision_-msgs/Detection3DArray* for further processing. An example of lidar detection visualization is shown in figure 6.

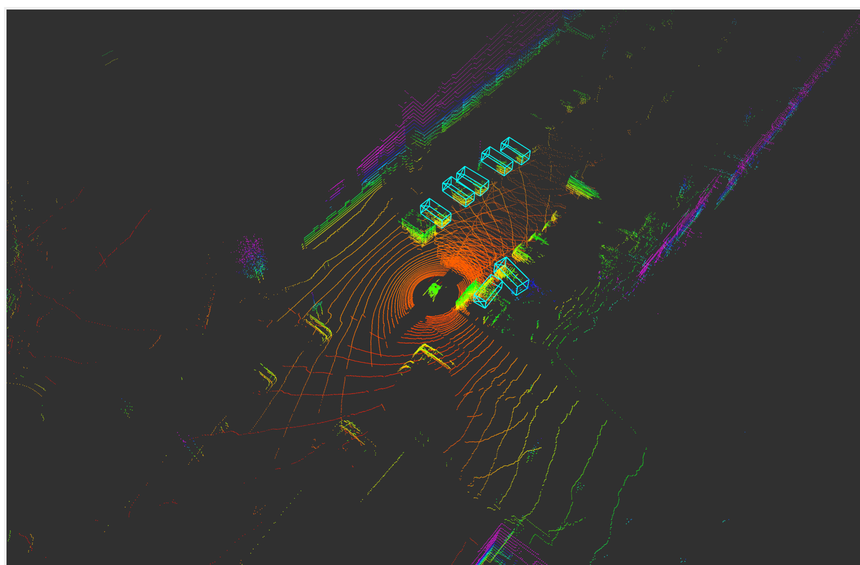


Fig. 6: Lidar detections projected into the local coordinate system. The PointPillars encoder implemented in the OpenPCDet framework has been used. The model used has been trained on the KITTI dataset.

8 Camera Detections

For the RGB camera-based detections, we've employed the YOLO V8 model [4]. YOLO V8 is the latest iteration of a model originally published by Redmon et al. in 2015 [15]. Since that, it has evolved and 5 different pre-trained model sizes for object detection and even segmentation are now available. This work [19] nicely summarizes the differences between each YOLO version. YOLO V8

models are trained on 80 different classes, but we only use a small subset of them, sometimes binning multiple classes into a single one. Our subset contains vehicles (all sorts), pedestrians, cyclists, traffic signs and animals, but any other configuration can be quickly set up to support different robotic tasks.

The Brno Urban Dataset contains 4 RGB cameras at 10 frames per second, so we've created a Python ROS2 node that runs a YOLO V8 nano segmentation model for a single camera. This node, just like the lidar detection node has two outputs. One output is purely for visualization in the form of *sensor_msgs/Image* and the second one for further processing is of type *vision_msgs/Detection2DArray*. A screenshot from our dataset with the output of the YOLO V8 nano segmentation model can be seen in figure 7.

These 2D detections can be then projected into the coordinate system of the aggregated lidar scan and from the points that lie inside of the segmented area, the median depth is calculated. With this information we can transfer our 2D detection into the 3D bounding box, the same representation as the lidar detections, so they can be matched together to create a more robust one.

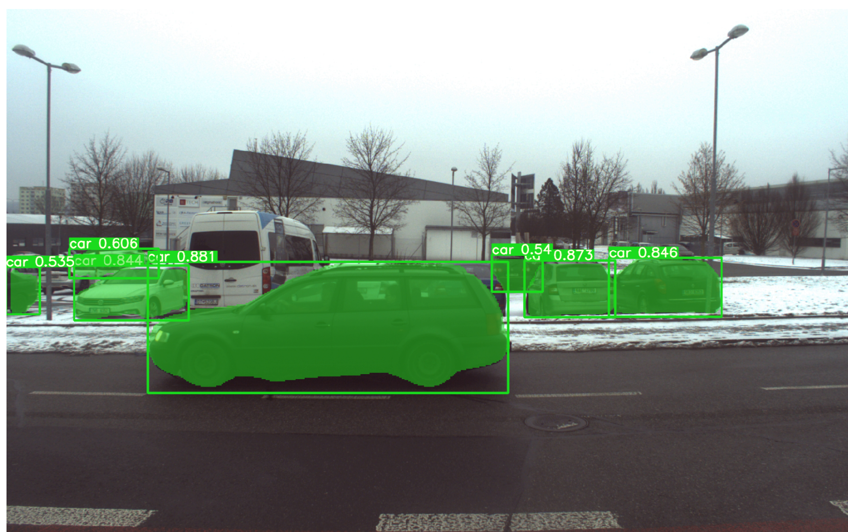


Fig. 7: Segmentation of the onboard camera done using YOLO V8 nano segmentation model. We're running this 4 times, once for every camera on the vehicle, in real-time.

9 Detection Fusion

As we already have, after reprojecting the RGB camera-based detections, data of an identical modality, we can compare it or fuse it easily. Currently, we employ

a basic intersection-over-union (IOU) metric to know, whether two detections overlap. If they do and the overlap is over a set threshold, the two (or more) detections get fused.

To comment on the detection reliability, we observe that the detection itself is much more reliable in the RGB domain. However, reprojecting the 2D detection into the 3D space is troublesome as the depth estimation is not always reliable when the detected object is partly obstructed. Detections in the lidar modality are a bit jumpy, but we think that if we've fine-tuned the pre-trained model to our data, the detections would get considerably better.

In the near future, we would like to implement an algorithm to track those fused detections between frames.

10 Instrumentation Layer

As this framework has been created mainly for testing new ideas and algorithms, we need a reliable way of measuring their runtime performance no matter which device they're currently running on. One of our requirements has been that this instrumentation is lightweight and doesn't have a noticeable impact on the performance while providing the information in real-time as the processing happens. To leverage the tool we are already using, ROS2, we've implemented an abstraction layer below every ROS2 node called *AtlasBaseNode*. This simple abstraction provides every node in our framework with common utility information like topic names from the config file and two timer functions.

Calls to the functions *StartTimer(const std::string timerName)* and *EndTimer(const std::string timerName)* can be inserted anywhere in the code where we want to measure runtime performance. Multiple running timers are also possible, each having a unique name. On *EndTimer()* call, the Node publishes *std_msgs::Float32* message onto a */instrumentation/timer/{timerName}* topic. This can be then inspected in multiple ways, but our preferred is using the PlotJuggler tool [2]. This allows us to plot multiple time series in real-time and even filtering or averaging the incoming data. This way it is easy to gain basic insight into the system and quickly iterate with solutions to bottlenecks.

As ROS2 has multiple layers and we can really only use this to instrument the application built on top of the ROS2 client library, there is a need for a tool that is able to measure the time between the publisher publishing the message and the subscriber's callback invoking. For these tracing needs the *ros2_tracing* [1] framework is really useful. Its main disadvantage is that it cannot run in real-time and the analysis of trace files can take some time, but there's really no way around this problem. With this framework, one can also gain more insight into how much system memory is the algorithm using and if there are any memory leaks.

11 Conclusion

For the use in our day-to-day testing and optimization of sensor fusion algorithms we've created a framework built on top of ROS2, that is able to process a practical number of sensor data in real-time. This framework contains a very comprehensive set of sensor-fusion algorithms that act as a backbone for the algorithms currently in the testing process. The pipeline is heavily reconfigurable, allowing for the use of multiple data sources, ROS2 bags, or even live sensor data directly from the driver. The framework is built in a way that allows for running parts of the system on different physical devices to split the workload. It also contains an instrumentation layer that allows for real-time measurement of runtime performance for quick evaluation. Though we've developed, tested and used our framework on the Brno Urban Dataset which is a dataset collected from a moving vehicle, it can be used for any type of robotic application.

Acknowledgement. The work has been performed in the project AI4CSM: Automotive Intelligence for/at Connected Shared Mobility No 101007326/8A21013. The work was co-funded by grants of Ministry of Education, Youth and Sports of the Czech Republic and Electronic Component Systems for European Leadership Joint Undertaking (ECSEL JU). The work was supported by the infrastructure of RICAIP that has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 857306 and from Ministry of Education, Youth and Sports under OP RDE grant agreement No CZ.02.1.01/0.0/0.0/17_043/0010085.

References

1. Bédard, C., Lütkebohle, I., Dagenais, M.: ros2_tracing: Multipurpose Low-Overhead Framework for Real-Time Tracing of ROS 2. IEEE Robotics and Automation Letters **7**(3), 6511–6518 (Jul 2022). https://doi.org/10.1109/LRA.2022.3174346
2. Faconti, D.: PlotJuggler 3.7 (Jul 2023), `https://github.com/facontidavide/PlotJuggler`, original-date: 2016-03-01T21:05:42Z
3. Geiger, A., Lenz, P., Urtasun, R.: Are we ready for autonomous driving? The KITTI vision benchmark suite. In: 2012 IEEE Conference on Computer Vision and Pattern Recognition. pp. 3354–3361 (Jun 2012). https://doi.org/10.1109/CVPR.2012.6248074
4. Jocher, G., Chaurasia, A., Qiu, J.: YOLO by Ultralytics (Jan 2023), `https://github.com/ultralytics/ultralytics`
5. Lang, A.H., Vora, S., Caesar, H., Zhou, L., Yang, J., Beijbom, O.: PointPillars: Fast Encoders for Object Detection From Point Clouds. In: 2019 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR). pp. 12689–12697. IEEE, Long Beach, CA, USA (Jun 2019). https://doi.org/10.1109/CVPR.2019.01298, `https://ieeexplore.ieee.org/document/8954311/`
6. Ligocki, A., Jelinek, A., Zalud, L.: Brno Urban Dataset - The New Data for Self-Driving Agents and Mapping Tasks. In: 2020 IEEE International Conference on Robotics and Automation (ICRA). pp. 3284–3290 (May 2020). https://doi.org/10.1109/ICRA40945.2020.9197277

7. Ligocki, A., Jelinek, A., Zalud, L.: Brno urban dataset: Winter extension. Data in Brief **40**, 107667 (Feb 2022). https://doi.org/10.1016/j.dib.2021.107667, https://www.sciencedirect.com/science/article/pii/S2352340921009422
8. Ligocki, A., Jelínek, A., Žalud, L.: Atlas Fusion - Modern Framework for Autonomous Agent Sensor Data Fusion. In: 2022 ELEKTRO (ELEKTRO) (May 2022). https://doi.org/10.1109/ELEKTRO53996.2022.9803587
9. Macenski, S., Martín, F., White, R., Clavero, J.G.: The Marathon 2: A Navigation System. In: 2020 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS). pp. 2718–2725 (Oct 2020). https://doi.org/10.1109/IROS45743.2020.9341207
10. Macenski, S., Soragna, A., Carroll, M., Ge, Z.: Impact of ROS 2 Node Composition in Robotic Systems. IEEE Robotics and Automation Letters **8**(7), 3996–4003 (Jul 2023). https://doi.org/10.1109/LRA.2023.3279614
11. Macenski, S., Tsai, D., Feinberg, M.: Spatio-temporal voxel layer: A view on robot perception for the dynamic world. International Journal of Advanced Robotic Systems **17**(2), 1729881420910530 (Mar 2020). https://doi.org/10.1177/1729881420910530, https://doi.org/10.1177/1729881420910530
12. Macenski, S., Foote, T., Gerkey, B., Lalancette, C., Woodall, W.: Robot operating system 2: Design, architecture, and uses in the wild. Science Robotics **7**(66), eabm6074 (2022). https://doi.org/10.1126/scirobotics.abm6074, https://www.science.org/doi/abs/10.1126/scirobotics.abm6074
13. Merriaux, P., Dupuis, Y., Boutteau, R., Vasseur, P., Savatier, X.: LiDAR point clouds correction acquired from a moving car based on CAN-bus data (Jun 2017), https://doi.org/10.48550/arXiv.1706.05886
14. Moore, T., Stouch, D.: A Generalized Extended Kalman Filter Implementation for the Robot Operating System. In: Menegatti, E., Michael, N., Berns, K., Yamaguchi, H. (eds.) Intelligent Autonomous Systems 13, vol. 302, pp. 335–348. Springer International Publishing, Cham (2016). https://doi.org/10.1007/978-3-319-08338-4_25, https://link.springer.com/10.1007/978-3-319-08338-4_25
15. Redmon, J., Divvala, S., Girshick, R., Farhadi, A.: You Only Look Once: Unified, Real-Time Object Detection. In: 2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR). pp. 779–788. IEEE, Las Vegas, NV, USA (Jun 2016). https://doi.org/10.1109/CVPR.2016.91, http://ieeexplore.ieee.org/document/7780460/
16. Rusu, R.B., Cousins, S.: 3D is here: Point Cloud Library (PCL). In: 2011 IEEE International Conference on Robotics and Automation. pp. 1–4. IEEE, Shanghai, China (May 2011). https://doi.org/10.1109/ICRA.2011.5980567, https://ieeexplore.ieee.org/document/5980567/
17. Team, A.D.: Autoware - the world's leading open-source software project for autonomous driving (Jul 2023), https://github.com/autowarefoundation/autoware
18. Team, O.D.: Openpcdet: An open-source toolbox for 3d object detection from point clouds. https://github.com/open-mmlab/OpenPCDet (2020)
19. Terven, J., Cordova-Esparza, D.: A Comprehensive Review of YOLO: From YOLOv1 and Beyond (Jun 2023), http://arxiv.org/abs/2304.00501
20. Zhang, B., Zhang, X., Wei, B., Qi, C.: A Point Cloud Distortion Removing and Mapping Algorithm based on Lidar and IMU UKF Fusion. In: 2019 IEEE/ASME International Conference on Advanced Intelligent Mechatronics (AIM). pp. 966–971 (Jul 2019). https://doi.org/10.1109/AIM.2019.8868647