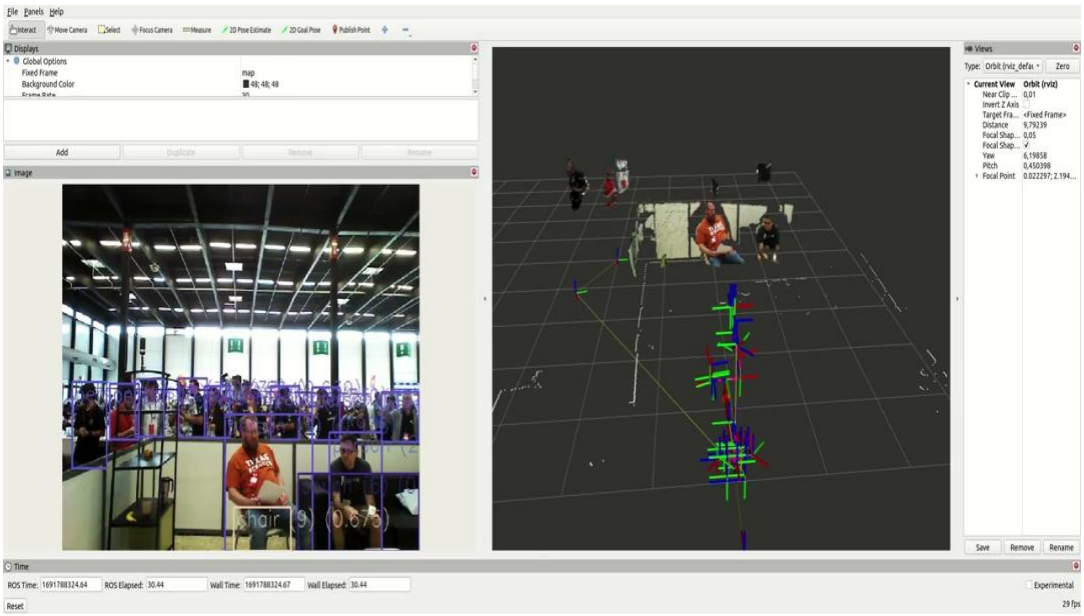
Real-Time Object Detection and Position Estimation Using YOLO and ROS: Implementation and Analysis

Aiyan Raaid
araaid@mun.ca
Memorial University of Newfoundland
St. John's, NL, Canada

Lu Qiao
qiaol@mun.ca
Memorial University of Newfoundland
St. John's, NL, Canada



Abstract

This report presents the design and implementation of a real-time, modular object detection and visualization system that integrates the YOLOv8 deep learning framework with the ROS2 ecosystem. Live video frames are captured from a webcam and processed on a Windows machine using YOLOv8n for object detection. The detected coordinates are then sent via UDP to a ROS2 node running in WSL2, where they are visualized as 3D markers in RViz.

By decoupling detection and visualization through socket-based communication, the system enables cross-platform compatibility and modular development. This architecture is lightweight, scalable, and suitable for robotics applications running on resource-constrained hardware, providing a practical foundation for real-time perception and visualization tasks.

Keywords—YOLO, ROS, object detection, position estimation, real-time tracking.

1. Introduction

Perception is a fundamental requirement for robotic systems, enabling autonomous agents to interpret their surroundings, avoid obstacles, and interact intelligently with dynamic environments. Among perception tasks, object detection and position estimation are critical for mobile robots, surveillance systems, and human-robot interaction.

Recent advances in deep learning, especially with models like YOLOv8 [1], have enabled real-time object detection with high accuracy. Simultaneously, ROS2 [2] has emerged as a standard for developing modular and distributed robotics applications, while RViz [3] provides rich 3D visualization tools for debugging and analysis.

This project presents a hybrid architecture where YOLOv8 runs on a Windows environment while ROS2 operates on Ubuntu via WSL2. The decoupling of vision and visualization subsystems via UDP allows real-time, cross-platform deployment, even on machines with limited resources.

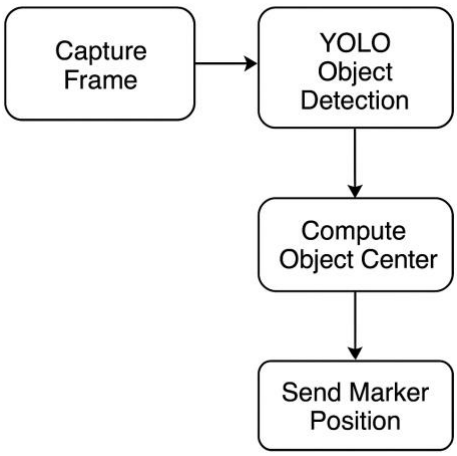
2. System Architecture

2.1 Overview

The system is composed of three decoupled components:

- Camera + YOLOv8 Detection (Windows): Captures real-time video from a laptop camera and detects objects using YOLOv8.
- UDP Listener + Marker Publisher (ROS2/WSL2): Receives coordinates from YOLO over UDP, creates ROS2 markers, and publishes them to a visualization topic.
- RViz Visualization: Displays detected objects in a 3D environment using spherical markers.

This modular structure enables independent development, debugging, and deployment of detection and visualization components.



2.2 Data Flow Pipeline

1. Webcam captures a frame on the Windows side.
2. YOLOv8 processes the frame and identifies objects with bounding boxes.
3. The center of each bounding box and its class label are extracted.
4. Coordinates and labels are formatted as strings and sent via UDP.
5. A ROS2 node on Ubuntu receives, decodes, and transforms the coordinates.
6. Visualization markers are published to /visualization_marker and rendered in RViz.

2.3 Mathematical Formulation

For each detected object, YOLO returns a bounding box: (x1,y1),(x2,y2)

The center (xc,yc) of the bounding box is computed as:

$$x_c = \frac{x_1 + x_2}{2}, \quad y_c = \frac{y_1 + y_2}{2}$$

These pixel coordinates are scaled to approximate spatial coordinates:

$$x = \frac{x_c}{1000}, \quad y = \frac{y_c}{1000}, \quad z = 0$$

TheROS2 visualization_msgs::Marker mess age uses this position to render a spherical marker.

2.4 Communication Protocol

A lightweight, connectionless UDP protocol was used to minimize latency. Each message follows the format:

<label>: (x_center, y_center)

person: (345.2, 288.1)

This approach ensures efficient and timely updates, tolerating minor packet loss in favor of speed.

2.5 YOLOv8 Detection

Using the ultralytics Python package:

```
from ultralytics import YOLO
model = YOLO('yolov8n.pt')
results = model(frame)
```

Results are parsed to extract class labels and box coordinates. Coordinates are formatted and transmitted:

```
message = f"{label}:
({center_x},{center_y})"
sock.sendto(message.encode(), addr)
```

2.6 ROS2 Marker Publisher

A Python ROS2 node receives and decodes UDP messages, constructs a spherical marker, and publishes it:

marker.pose.position = Point(x=x / 1000.0, y=y / 1000.0, z=0.0)

marker.type = Marker.SPHERE

marker.color.r = 1.0

marker.scale.x = marker.scale.y = marker.scale.z = 0.2

3. Implementation

3.1 Directory Structure

```
for result in results:
    for box in result.boxes.xyxy:
        center_x = (box[0] + box[2]) / 2
        center_y = (box[1] + box[3]) / 2
        label = result.names[int(box.cls)]
        message = f"{label}:
({center_x},{center_y})"
        sock.sendto(message.encode(), addr)
```

3.2 YOLO Sender

Processes frames and sends coordinates via UDP:

```
        center_x = (box[0] + box[2]) / 2

for result in results:
    for box in result.boxes.xyxy:
        center_x = (box[0] + box[2]) / 2
        center_y = (box[1] + box[3]) / 2
        label = result.names[int(box.cls)]
        message = f"{label}:
({center_x},{center_y})"
        sock.sendto(message.encode(), addr)
```

3.3 Marker Publisher (ROS2)

marker.pose.position = Point(x=self.latest_x / 1000.0,
y=self.latest_y / 1000.0, z=0.0)
marker.type = Marker.SPHERE
marker.scale.x = marker.scale.y = marker.scale.z = 0.2
marker.color.r = 1.0

3.4 Optional: camera_socket_receiver.py

Receives and displays image frames via sensor_msgs/Image, useful for debugging or RViz image overlay.

3.5 RViz Configuration

- Fixed Frame: base_link
- Topic: /visualization_marker
- Marker Type: Sphere
- Size: 0.2

4. Results

4.1 Real-Time Visualization

- Successfully detected and visualized objects like person, bottle, and cell phone.
- Red spherical markers follow detected objects in RViz.
- System is responsive to motion and updates smoothly.

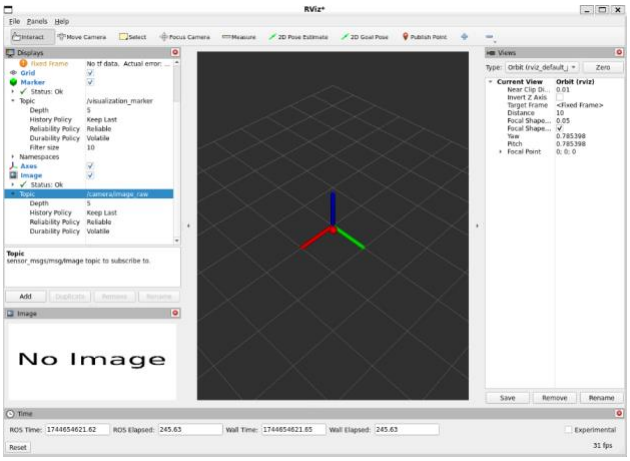


Figure 1: without Image

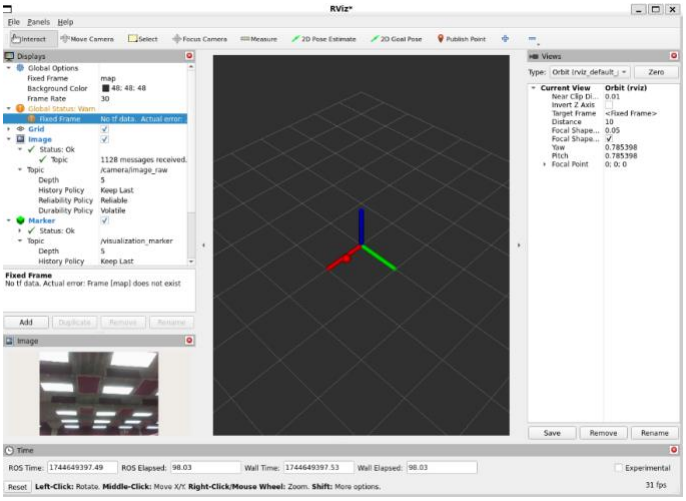


Figure 2: with Image

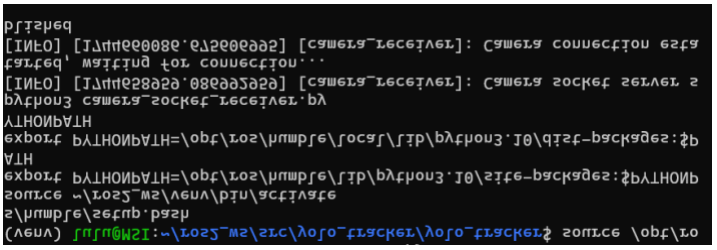


Figure 3: How to run

camera_socket_receiver.py

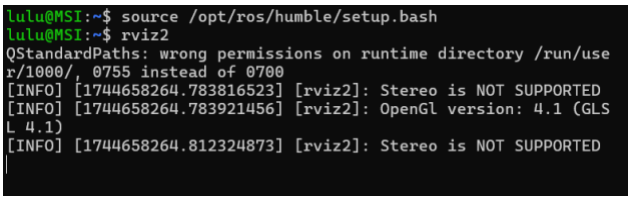


Figure 4: How to run rviz2

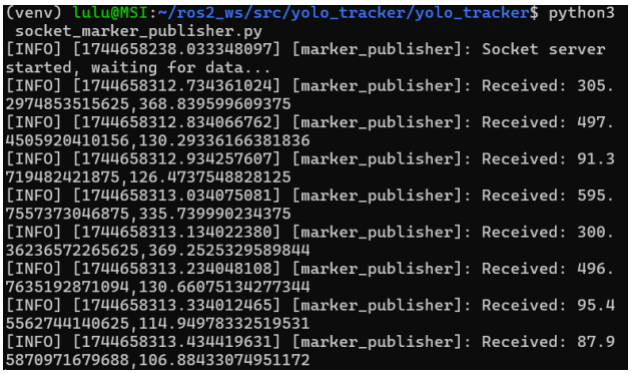


Figure 5: How to run

socket_marker_publisher.py
Appendix: Sample Output

Sent: person: (345.2, 288.1)

- Red dot follows the moving object.
- Markers update in real-time.

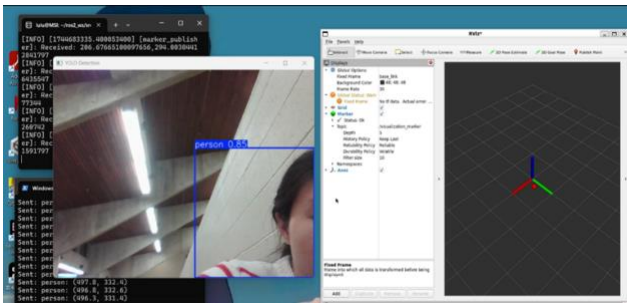


Figure 6: Successfully visualizes moving objects with red spherical markers in RViz.

4.2 Performance Metrics

Component	Performance
YOLOv8 Inference	~15 FPS
UDP Transmission	1–2 ms / packet
Marker Update (ROS2)	~10 Hz
Total Visualization Lag	~3 seconds (observable)

4.3 Limitations

• **Marker Flickering in RViz under high load**
During high-frequency updates or when tracking multiple objects simultaneously, RViz may experience rendering inconsistencies. This results in flickering or momentary disappearance of markers, likely due to message queue overflows or update conflicts, especially when QoS settings are not optimized.

• **Latency (~3s) observed during complex movements**
Although the system is designed for real-time performance, noticeable latency—up to 3 seconds—can occur when detecting fast or erratic object motion. This may be due to cumulative delays from YOLO inference, socket communication, and RViz rendering, particularly on resource-constrained hardware.

• **No persistent tracking — each frame treated independently**
The current setup does not maintain object identity across frames. Each detection is processed in isolation, meaning the system cannot distinguish whether an object in frame *t+1* is the same as in frame *t*. This limits the application's ability to perform trajectory tracking or analyze movement patterns over time.

• **Depth estimation not included — limited to 2D XY marker display**
The system currently computes only the x and y pixel center of bounding boxes, which are scaled for marker placement in 2D space. Without depth information (z-axis), the system lacks spatial depth perception, reducing its suitability for applications that require full 3D awareness such as grasp planning or obstacle mapping.

5. Conclusion

This system successfully integrates state-of-the-art deep learning for object detection with the ROS2 middleware to achieve real-time, cross-platform visualization of object positions. By offloading object detection to a Windows-based YOLOv8 model and transmitting positional data to ROS2 via UDP, the architecture maintains a clear separation of responsibilities while minimizing latency. This design also promotes modularity, allowing each component to be developed and debugged independently—an essential quality for scalable robotics systems.

The use of red spherical markers in RViz enables clear and immediate visualization of detected object positions, which is valuable

for tasks such as navigation, object handoff, and behavioral analysis in human-robot interaction. Although the system currently provides only 2D positional data and exhibits some latency during high-speed movement, it lays a robust groundwork for real-time robotic perception pipelines. It also demonstrates the feasibility of deploying advanced AI perception modules even on limited consumer-grade hardware.

Future Work

- **Integrate Kalman Filters or Deep SORT [4] for temporal tracking**
Currently, the system performs frame-by-frame detection without maintaining object identities over time. Implementing Kalman Filters or Deep SORT would allow for continuous object tracking, enabling smoother motion estimation, handling occlusions, and supporting trajectory prediction—essential for tasks like autonomous navigation or gesture recognition.

- **Extend to 3D by incorporating stereo vision or depth cameras**
The current implementation estimates position using 2D image coordinates. Integrating stereo vision systems or depth sensors like Intel RealSense would enable accurate 3D localization. This would allow the system to estimate depth (z-axis), enhancing its utility for applications requiring full spatial awareness, such as robotic arm manipulation or obstacle avoidance.

- **Replace UDP with ROS2-native publishers for seamless integration**
While UDP allows low-latency communication between Windows and ROS2 (on WSL2), transitioning to ROS2-native publishing would eliminate the need for custom socket handling. This would allow direct communication between ROS2 nodes, improving reliability, synchronization, and making it easier to integrate with existing ROS2 tools and packages.

- **Optimize visualization performance using ROS2 QoS settings**
ROS2 provides configurable Quality of Service (QoS) policies for tuning message delivery. By optimizing parameters like message reliability, history depth, and update frequency, the system can reduce flickering, prevent message loss under load, and maintain smoother marker updates in RViz—especially important when tracking multiple objects or working in unstable network conditions.

6. References

1. Redmon, J., et al. You Only Look Once: Unified, Real-Time Object Detection. CVPR, 2016.
2. ROS2 Documentation. https://docs.ros.org/en/humble
3. RViz Markers API. https://docs.ros2.org/latest/api/visualization_msgs/msg/Marker.html
4. Wojke, N., Bewley, A., Paulus, D., Simple Online and Realtime Tracking with a Deep Association Metric, ICIP, 2017.
5. Ultralytics YOLOv8. https://docs.ultralytics.com
6. Geiger, A., et al. Are we ready for Autonomous Driving? The KITTI Vision Benchmark Suite. CVPR, 2012.
7. Huang, J., et al. Speed/accuracy trade-offs for modern object detectors. CVPR, 2017.