Go, change the world®

ARTIFICIAL INTELLIGENCE IN AUTONOMOUS VEHICLES

VII semester

Department of AIML

RV College of Engineering

Course Incharge: Dr. Viswavardhan Reddy Karna

2024-2025

Course Contents – Unit - IV

Go, change the world®

Reinforcement Learning-Based Planning and Control:

- Introduction to Reinforcement Learning.
- Learning Based Planning and Control in Autonomous Driving

Client Systems for Autonomous Driving:

- Autonomous Driving: A Complex System
- Operating System for Autonomous Driving
- Computing Platform

Key Characteristics of Reinforcement Learning

- Learning is interactive with the environment
- Closed-loop learning process
- The main entity: Agent
- Everything outside the agent: Environment

Agent-Environment Interaction

- The agent makes decisions by taking actions
- The environment responds with new states and rewards
- Continuous iterative process

Figure 7.1: Reinforcement learning framework: agent interacting with environment by taking actions, sensing state, and getting rewards.

Reinforcement Learning-Based Planning and Control

Go, change the world®

The Learning Process

- Learning occurs in rounds indexed by time ($t = 0,1,2,3\ldots$)
- At each time step t:
  - The agent perceives the environment (State S_t)
  - Takes an action (A_t)
  - Receives a new state (S_{t+1}) and a reward (R_t)

Reward Function

- Defines how rewards are generated by the environment
- Maps state or state-action pairs to a scalar reward
- The agent aims to maximize cumulative rewards

Episodic vs. Continuing Tasks

- **Episodic Learning:**
  - Fixed sequences of states
  - Ends in terminal states
- **Continuing Tasks:**
  - Infinite sequence of states
  - Requires discounted reward computation

Return Computation

- Episodic tasks: Sum of rewards
- Continuing tasks: Discounted rewards
- Return function:
  - $G_t = R_{t+1} + \gamma R_{t+2} + \gamma 2 R_{t+3} \ldots$
  - γ (gamma) is the discount factor ($0 \leq \gamma \leq 1$)

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} \ldots = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1} ,$$

Policy and Decision Making

- Policy (π): Strategy for decision making
- Maps states to actions
- Can be:
    - Simple (lookup table)
    - Complex (deep neural network)

Value Functions

- Measures the expected return for being in a state
- State-Value Function: $V_{\pi(s)} = E_{\pi} (G_t \mid S_t = s)$
- State-Action Value Function (Q-value): $Q_{\pi(s,a)}$

in Chapter 6) where the state transitions are Markov, the value function could be written as: $V_\pi(s) = E_\pi\ (G_t \mid S_t = s) = E_\pi\ (\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \mid S_t = s)$, where E_π is the expectation operator given the policy π. This value function is also called the *state-value function for policy* π. Similarly, the Q-value function is the mapping of state-action pairs to a scalar value representing the expected return at state S_t after taking action A_t and following the policy π afterward. It is denoted as: $Q_\pi\ (s,a) = E_\pi\ (G_t \mid S_t = s, A_t = a) = E_\pi\ (\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \mid S_t = s, A_t = a)$.

- Q-Learning is one of the most popular algorithms used for reinforcement learning.

- The idea is to learn to approximate the strategy that maximizes the expected return at any given state s_t by taking action a_t and then following the optimal strategy: $Q(s_t, a_t) = \max_\pi R_{t+1}$.

- How we then select the policy is based on: $\pi(s_t) = \text{argmax}_{a_t} Q(s_t, a_t)$.

- The key problem in Q-learning is how to accurately estimate the Q-function that maps state-action pairs to an optimal expected return

Reinforcement Learning-Based Planning and Control

- Q-Learning is one of the most popular algorithms used for reinforcement learning.

- The idea is to learn to approximate the strategy that maximizes the expected return at any given state s_t by taking action a_t and then following the optimal strategy: $Q(s_t, a_t) = \max_\pi R_{t+1}$.

- How we then select the policy is based on: $\pi(s_t) = \operatorname{argmax}_{a_t} Q(s_t, a_t)$.

- The key problem in Q-learning is how to accurately estimate the Q-function that maps state-action pairs to an optimal expected return

Q-Learning algorithm

function Q_Leaning(*Episodes*)
    Initialize the $Q(s,a)$, and $Q(terminate\text{-}state,:)=0$
    for each episode in *Episodes*:
        Initialize to start state S
        repeat (**for each** *step* of *episode*):
            choose an **action** $a \in A$ using the policy from the Q table with ε -greedy
            Take the action a and observe the reward R and the next state s'
            $Q(s,a) \leftarrow Q(s,a) + \alpha[R + \gamma \max_{a'} Q(s',a') - Q(s,a)]$
            $S \leftarrow s'$
        until S is a terminal state

Deep-Q-Learning algorithm.

function DQN_Leaning(*Episodes*)
  Initialize the $Q(s,a)$ with random weights
  Initialize the replay memory D
  for each episode in *Episodes*:
    repeat :
      choose an **action** $a \in A$ using the policy from the Q table with ε -greedy
      take the action a and observe the reward r and the next state s'
      store experience (s, a, r, s') in replay memory D
      sample random transition (ss, aa, rr, ss') from replay memory D
      calculate target for each minibatch transition
        if is terminal state:
          $tt = rr$
        else:
          $tt = rr + \gamma \max_{a'} Q(s', a') - Q(s, a)$
        train deep neural network Q with $Loss = {}^{1}/_{2}\,(tt - Q(ss, aa))^{2}$
      $s \leftarrow s'$
    until s is a terminal state

The neural network **Q(s, a)** is trained using a squared loss function:

$$Loss = \frac{1}{2}(tt - Q(ss, aa))^2$$

This updates the network weights to make `Q(s, a)` closer to `tt`.

Reinforcement Learning-Based Planning and Control

Deep-Q-Learning algorithm.

```
function DQN_Leaning(Episodes)
    Initialize the Q(s,a) with random weights
    Initialize the replay memory D
    for each episode in Episodes:
        repeat :
            choose an action a ∈ A using the policy from the Q table with ε -greedy
            take the action a and observe the reward r and the next state s'
            store experience (s, a, r, s') in replay memory D
            sample random transition (ss, aa, rr, ss') from replay memory D
            calculate target for each minibatch transition
                if is terminal state:
                    tt = rr
                else:
                    tt = rr + γ max_a' Q(s', a') − Q(s, a)
                train deep neural network Q with Loss = 1/2 (tt − Q(ss, aa))^2
            s ← s'
        until s is a terminal state
```

- If `s'` is a **terminal state**, the target Q-value `tt` is simply the observed reward (`rr`).
- Otherwise, the **Bellman equation** is used:

$$tt = r + \gamma \max_{a'} Q(s', a')$$

  - `γ` (gamma) is the **discount factor** ($0 \leq \gamma \leq 1$).
  - `max Q(s', a')` is the highest predicted Q-value for the next state.
  - The difference `tt - Q(s, a)` is the **TD (Temporal Difference) error**.

Deep-Q-Learning algorithm.

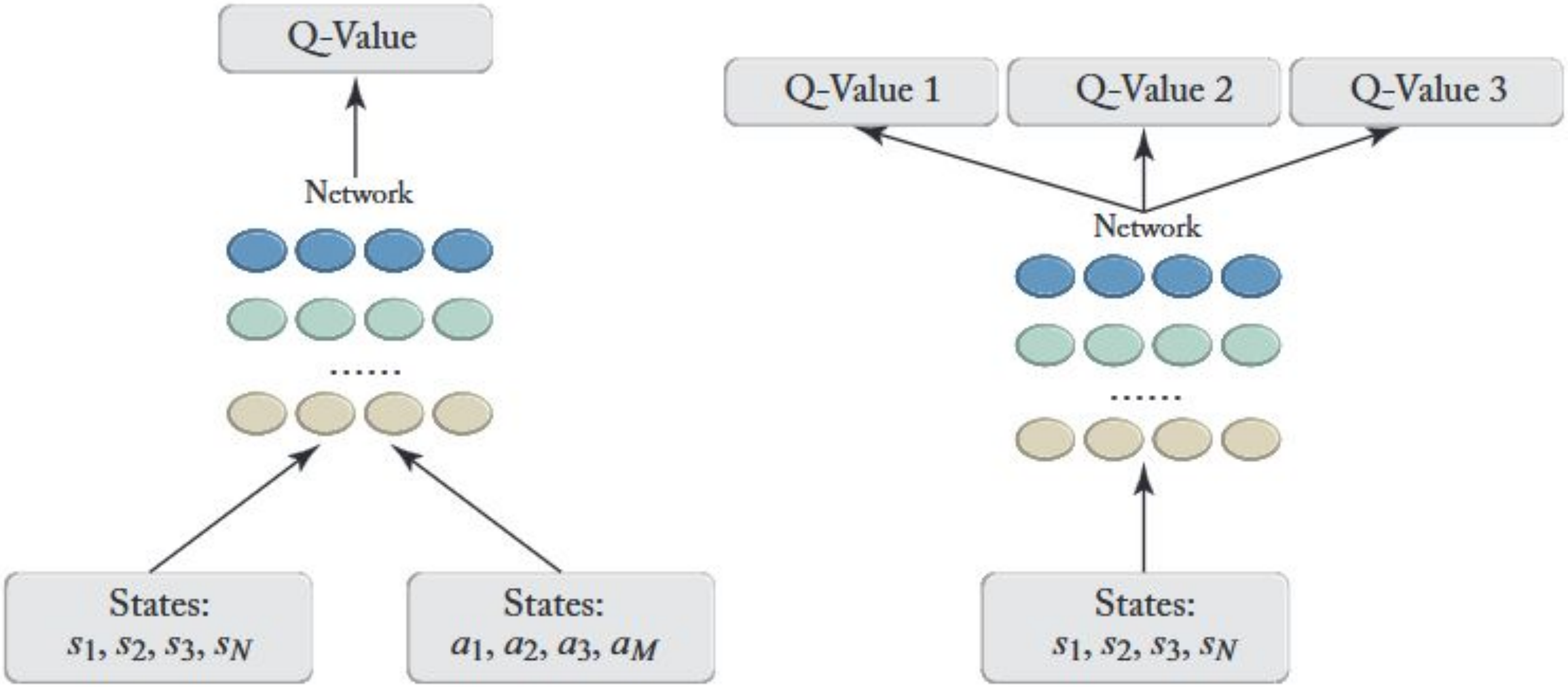


Figure 7.3: Deep-Q-Learning (DQN): Neural network structures.

ACTOR-CRITIC METHODS

- Asynchronous Advantage Actor-Critic (A3C) algorithm presented by Google's DeepMind.
- It has been shown to be faster, simpler, and more robust than the traditional Deep-Q-Learning algorithms on standard reinforcement learning tasks
- The structure of an A3C network differs from DQN by using multiple learning agents instead of one.
- Each agent has its own neural network and interacts with its own environment.
- A global network aggregates updates from worker agents.
- Asynchronous learning: Each agent updates its policies independently before synchronizing with the global network.

A3C Training Workflow

.Initialize a global network and reset each individual worker network to the global network.

.Each individual worker agent interacts and learns within its own environment.

.Each individual worker agent computes the loss function for its neural network.

.Each individual worker agent updates its gradient from the computed loss.

.Worker agents together update the global network with the appropriate gradients.

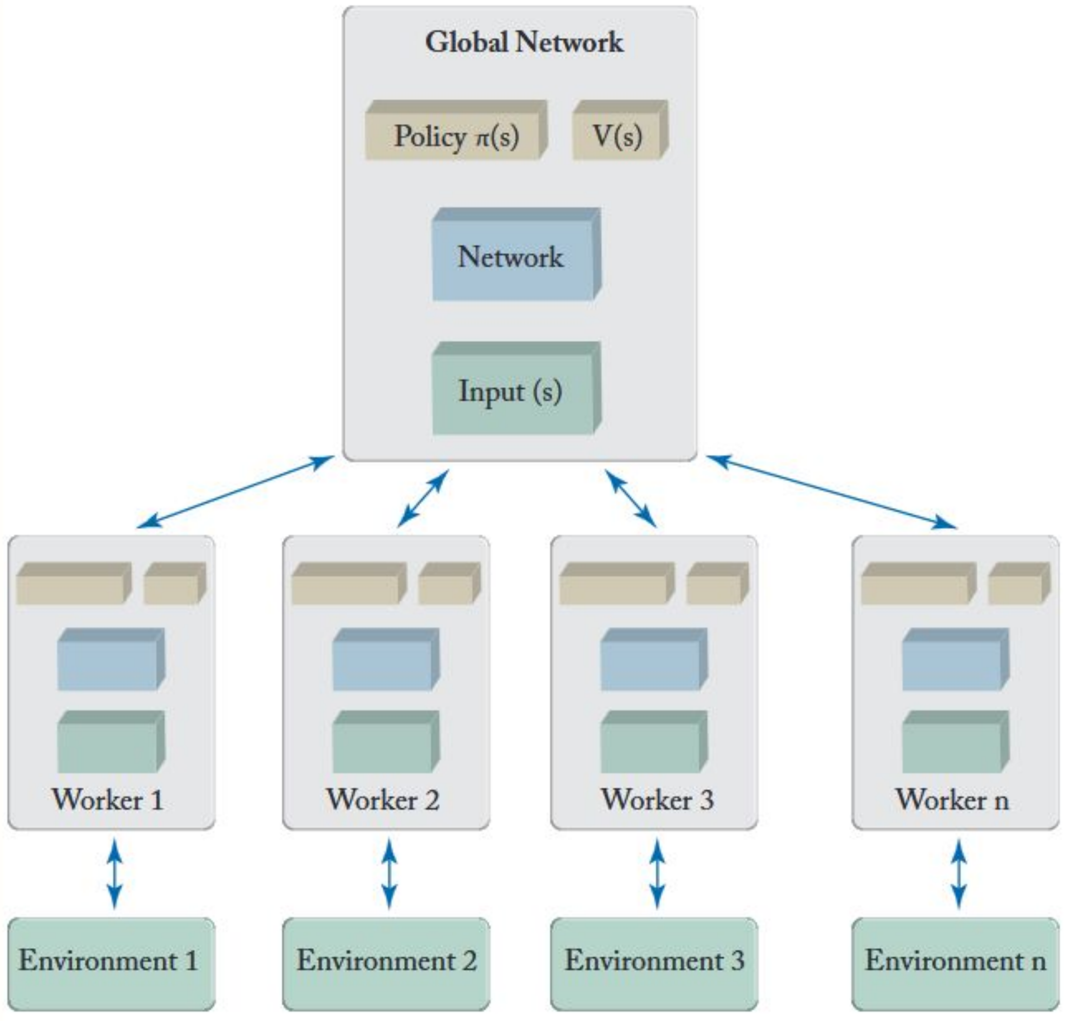


Figure 7.6: Asynchronous Advantage Actor-Critic Framework (based on [11]).

Reinforcement Learning-Based Planning and Control

Go, change the world®

Shared Network Architecture

- Both value and policy networks share:
  - **Convolutional Layers** for spatial feature extraction.
  - **LSTM Layer** for handling temporal dependencies.
- Helps maintain location invariance and sequence learning.

Actor-Critic Mechanism

- **Critic (Value Function $V\pi$ (s))**:
  - Evaluates the quality of a state.
  - Guides policy updates.
- **Actor (Policy π (s))**:
  - Determines the optimal action.
  - Adjusts based on critic feedback.

Advantage Function

- Measures improvement over expected return:
  - **$A(s, a) = Q(s, a) - V(s)$**
  - Helps refine policy updates.
- Advantage function ensures more effective training.

Loss Functions

- **Value Loss:**
  - $L\_value = \sum (R - V(s))^2$
- **Policy Loss:**
  - $L\_policy = A(s) * \log(\pi(s)) + H\pi * \beta$
- Includes entropy term $H\pi$ for balancing exploration and exploitation.

Exploration vs. Exploitation

- **Entropy ($H\pi$) measures policy uncertainty.**
  - High entropy → More exploration.
  - Low entropy → More exploitation.
- Helps stabilize learning and avoid local optima.

REINFORCEMENT LEARNING ON BEHAVIORAL DECISION

- The main goal of applying reinforcement learning (RL) in behavioral decisions is to manage diverse traffic scenarios.
- Strict adherence to traffic rules is not always effective in real-world driving.
- "Long-tail" cases in behavioral decisions require adaptive decision-making.
- Human driving experiences can serve as valuable examples for RL-based systems.
- RL can complement rule-based behavioral decision approaches, which remain the industry standard.

REINFORCEMENT LEARNING ON BEHAVIORAL DECISION

RL-Based Behavioral Decision Model

•The action space (Desires) is defined as:

$$\mathbf{D = [0, v_{max}] \times L \times \{g, t, o\}^n}$$

where:

- $\mathbf{v_{max}}$: Desired target speed of the vehicle
- **L**: Set of discrete lateral lane positions
- **g**: Give way (yield), **t**: Take (overtake)
- **o**: Keep an offset distance (nudge/attention)

REINFORCEMENT LEARNING ON BEHAVIORAL DECISION

State Space

- The environment model is generated by interpreting sensory information.
- Includes additional information such as kinematics of moving objects.

Implementation & Limitations

- RL agent initialized with imitation learning.
- Updated using iterative Policy Gradient approach.

REINFORCEMENT LEARNING ON PLANNING AND CONTROL

- The key challenge in RL-based planning and control is how to design the state spaces.
- To compute the motion planning or feedback control level actions, it is necessary to include both autonomous vehicle information and the surrounding environment.
- If we wont take sensory data – which is raw, the state spaces will somehow incorporate structured information about AV and its surroundings - large multi-dimensional continuous space.
- To tackle the challenge of a continuous state space, cell-mapping techniques can be combined with reinforcement learning

REINFORCEMENT LEARNING ON PLANNING AND CONTROL

Continuous State Space

- In reinforcement learning (RL), the **state space** represents all possible situations an agent can encounter.

- A **continuous state space** means the states are not discrete but can take on any value within a given range.

- Example: In autonomous driving, the position and speed of a car can take infinitely many values rather than being confined to a limited set of predefined states.

Cell-Mapping Techniques

- **Cell-mapping** is a method that divides a continuous state space into discrete cells. Each cell represents a group of similar states

REINFORCEMENT LEARNING ON PLANNING AND CONTROL

Combining Cell-Mapping with RL

- By discretizing the state space into manageable cells, RL algorithms can operate more effectively without requiring infinite memory or computational power.

- This hybrid approach is useful for problems where a fully continuous representation is too complex but full discretization is too simplistic.

- It can improve convergence speed and stability in RL applications like robotic control, traffic management, and autonomous systems.

LEARNING-BASED PLANNING AND CONTROL IN AUTONOMOUS DRIVING

REINFORCEMENT LEARNING ON PLANNING AND CONTROL

State Symbol	State Variable	Range
$X_1 = v$	Velocity	$-1.5 \leq X_1 \leq 1.5$ "m/s"
$X_2 = x$	X Cartesian Coordinate	$-0.9 \leq X_2 \leq 0.9$ "m"
$X_2 = y$	Y Cartesian Coordinate	$-1.3 \leq X_3 \leq 1.3$ "m"
$X_2 = \theta$	Orientation	$-\pi \leq X_4 \leq \pi$ "rad"

Action Symbol	Action Values
Voltage in traction motor	−18 V 0 V 18 V
Steering angle	−23° 0° 23°

Algorithm 1 CACM-RL

```
Initialise Q-Table(s,a) y Model_Table
x ← current state
s ← cell(x)
IF      s ∈ drain or s ∈ goal or s ∈ safety_area
THEN    F_reactive(x)
ELSE    IF      D-k-adjoining(x,x')
        THEN    Q-Table(s,a) ← s',r
                Model_Table ← IT(x,x')
        a ← policy(s)
        Execute action a on the vehicle
        Observe the new state x' and r
UNTIL the end of the learning stage
FOR all (s,a), repeat N times
  x̃' ← Model_Table, DT(x,x')
  s̃' ← cell(x̃')
  Q-Table(s,a) ← s̃', r
```

REINFORCEMENT LEARNING ON PLANNING AND CONTROL

Initialization Phase

1.Initialize Q-Table(s, a) and Model_Table

- The **Q-Table** stores the expected rewards for taking actions in different states.
- The **Model_Table** keeps track of the transitions and rewards for each state-action pair.

2.x ← current state

- The current state of the system (e.g., velocity, position, orientation) is assigned to x.

3.s ← cell(x)

- The continuous state x is mapped to a **cell** in the discretized state space. This step is crucial for handling continuous environments in reinforcement learning.

REINFORCEMENT LEARNING ON PLANNING AND CONTROL

State Classification & Action Selection Phase

4.IF s is drain OR s is goal OR s is safety_area

- If the agent reaches a **drain state** (undesirable), a **goal state** (successful outcome), or a **safety area**, then:

5.THEN F_reactive(x)

- Apply a reactive function to handle these special cases. This function could reset the agent or enforce constraints.

6.ELSE IF D-k-adjoining(x, x')

- Check if the new state x' is **D-k adjoining**, meaning it is within a defined neighborhood of the current state x.

7.THEN Q-Table(s, a) ← s', r

- Update the Q-Table by storing the new state s' and the reward r for taking action a in state s.

8.Model_Table ← (x, π(x'))

- Update the **Model_Table** with the new state transition (x, π(x')), where π(x') represents a transition function that determines the next state.

9.a ← policy(s)

- Select an action a based on the policy, which could be an **ε-greedy policy**, **softmax**, or another decision-making strategy.

REINFORCEMENT LEARNING ON PLANNING AND CONTROL

Action Execution & Learning Phase

10.Execute action a on the vehicle

- Apply the chosen action a to the system (e.g., adjust voltage or steering angle).

11.Observe the new state x' and r

- The agent receives feedback by observing the new state x' and the reward r from the environment.

12.UNTIL the end of the learning stage

- Repeat the process until the training phase is complete.

13.FOR ALL (s, a), repeat N times

- Perform updates for all state-action pairs multiple times to refine learning.

REINFORCEMENT LEARNING ON PLANNING AND CONTROL

- **x' ← Model_Table, DT(x', x')**
- Retrieve the next predicted state from the **Model_Table** using a **Dynamic Transition (DT) function**.

- **s' ← cell(x')**
- Convert the continuous state x' into its corresponding cell representation.

- **Q-Table(s, a) ← g', r'**
- Final Q-Table update with the reward r' and new state g'.

Client Systems for Autonomous Driving

- Autonomous driving is an integration of multiple technologies.

- The client system consists of:
  - **Operating system and Hardware platform**

- Coordination between these components is essential for performance.

Hardware Platform Overview

- Follows the **sensing-perception-action** paradigm.

- **Sensors** collect environmental data.

- **Computing platform** processes data for perception and action computation.

- **Control platform** executes the action plans.

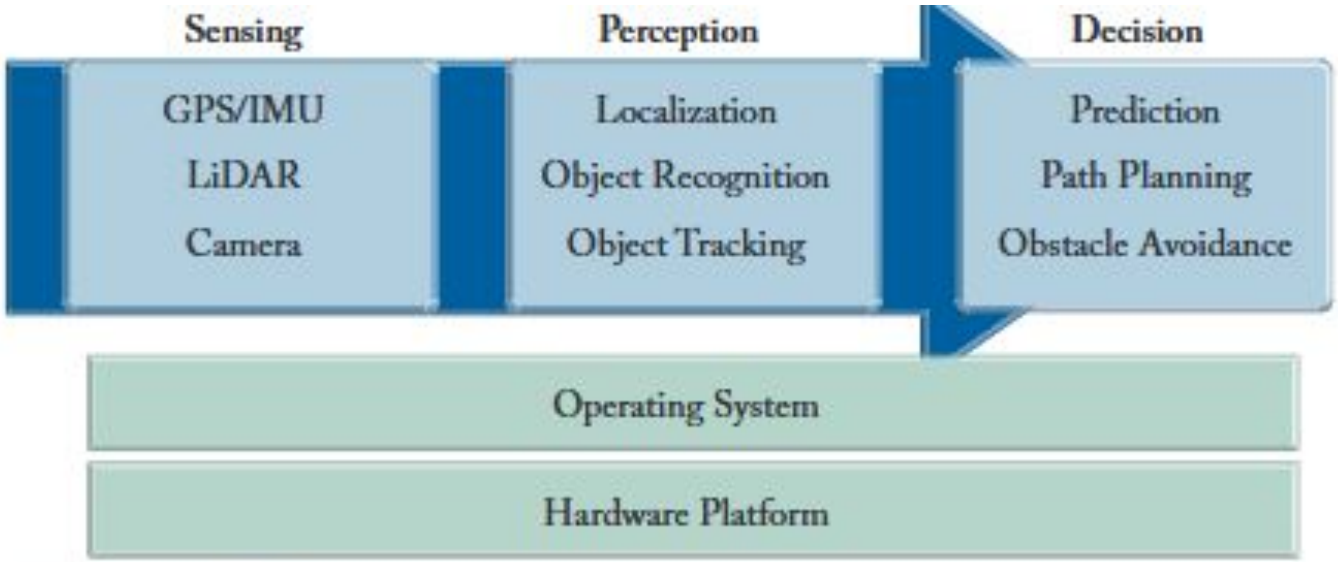


Figure 8.1: Hardware platform for autonomous driving.

Client Systems for Autonomous Driving

Go, change the world®

Role of the Operating System

- Manages communication between hardware components.
- Coordinates **resource allocation** for real-time tasks.

Real-Time Processing Requirements

- Example: **Camera requires 60 frames per second**.
- Each frame must be processed in **less than 16 ms**.
- Increased data load affects processing efficiency.

Challenges in Resource Allocation

- **Data bursts** (e.g., LiDAR point clouds) can overload CPU.
- Can lead to **dropped frames** from the camera.
- Need a mechanism to restrict resource usage per component.

Client Systems for Autonomous Driving

Go, change the world®

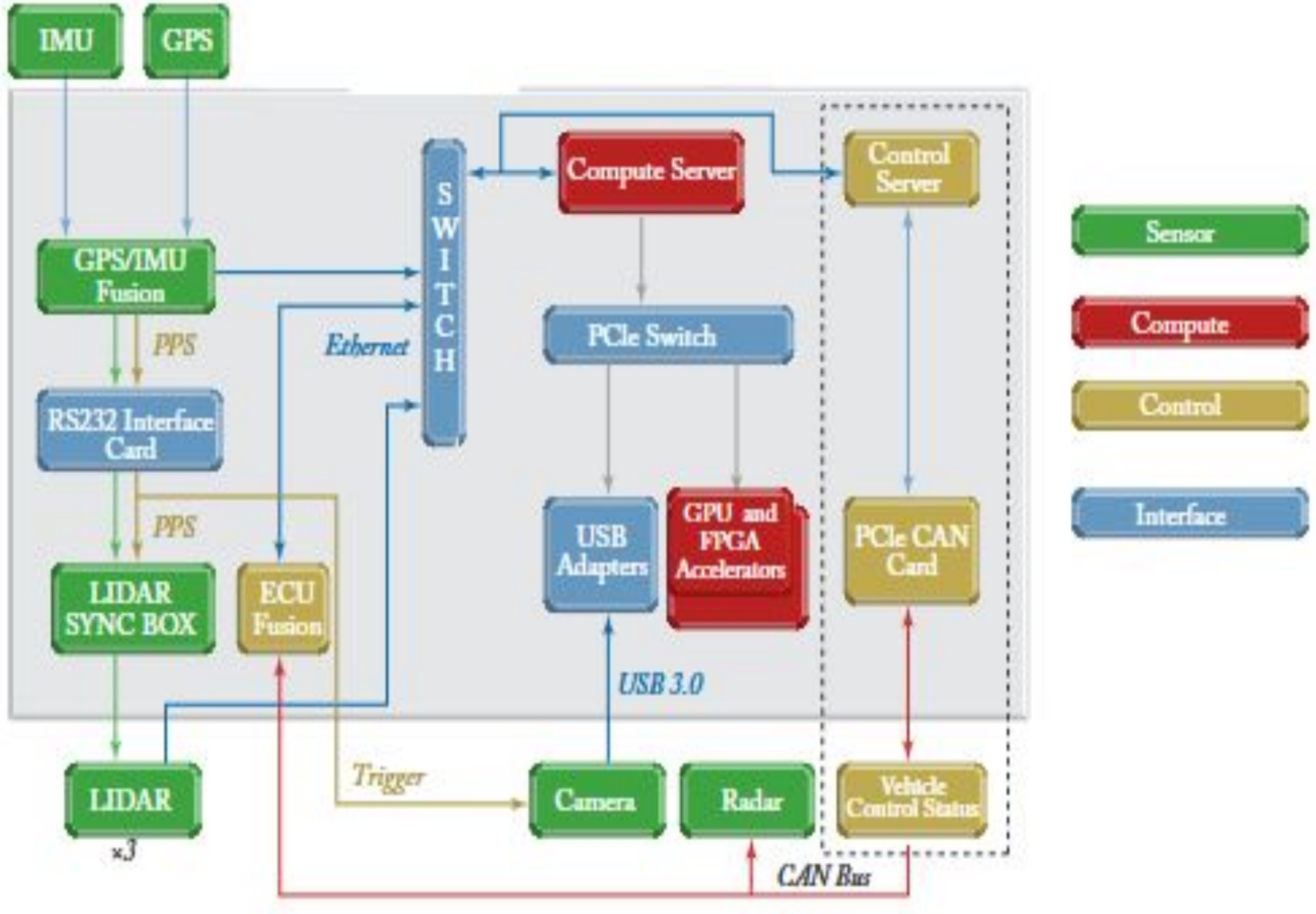


Figure 8.2: Hardware platform for autonomous driving.

Client Systems for Autonomous Driving

- The two main functions provided by the operating system include communication, and resource allocation.

- The main components are divided into 3 types: ROS Master, ROS Node, and ROS Service.

- Function of ROS master is to provide name service.

- It stores the operating parameters that are required at startup, the name of the connection between the upstream node and the downstream node, and the name of the existing ROS services.

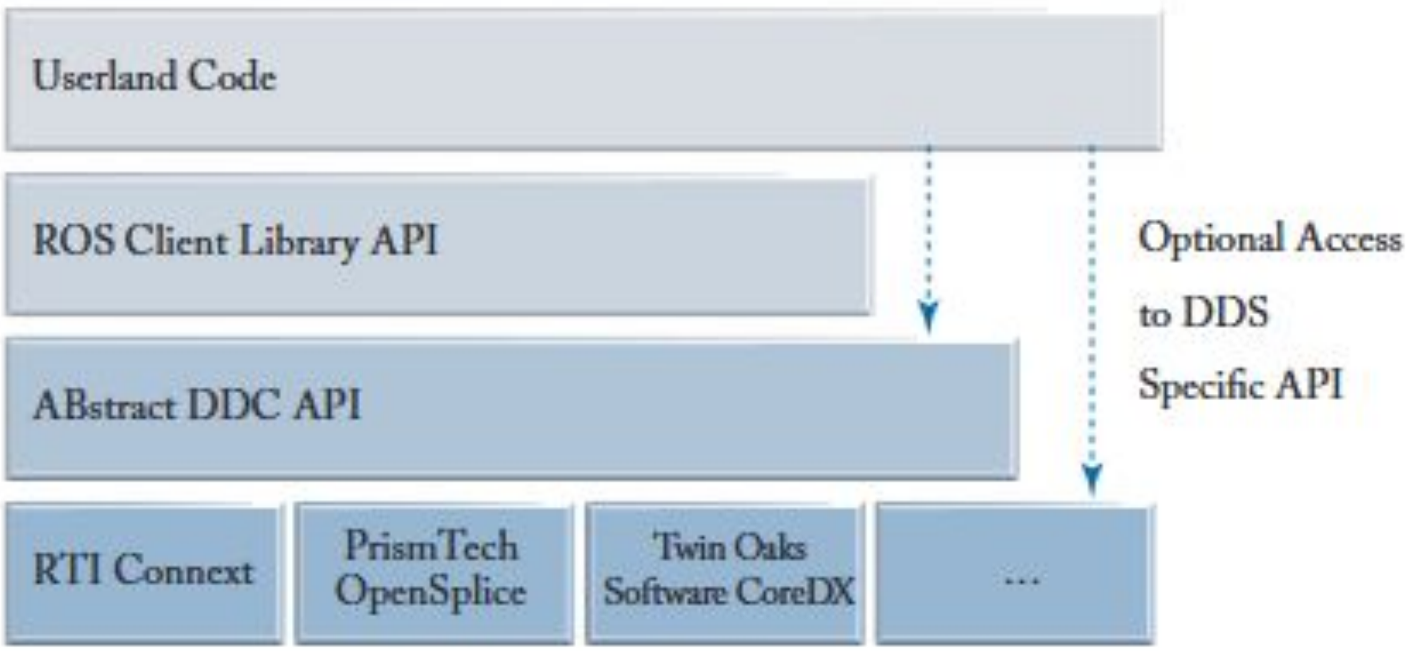


Figure 8.3: ROS 2.0 communication with DDS.

Client Systems for Autonomous Driving

Go, change the world®

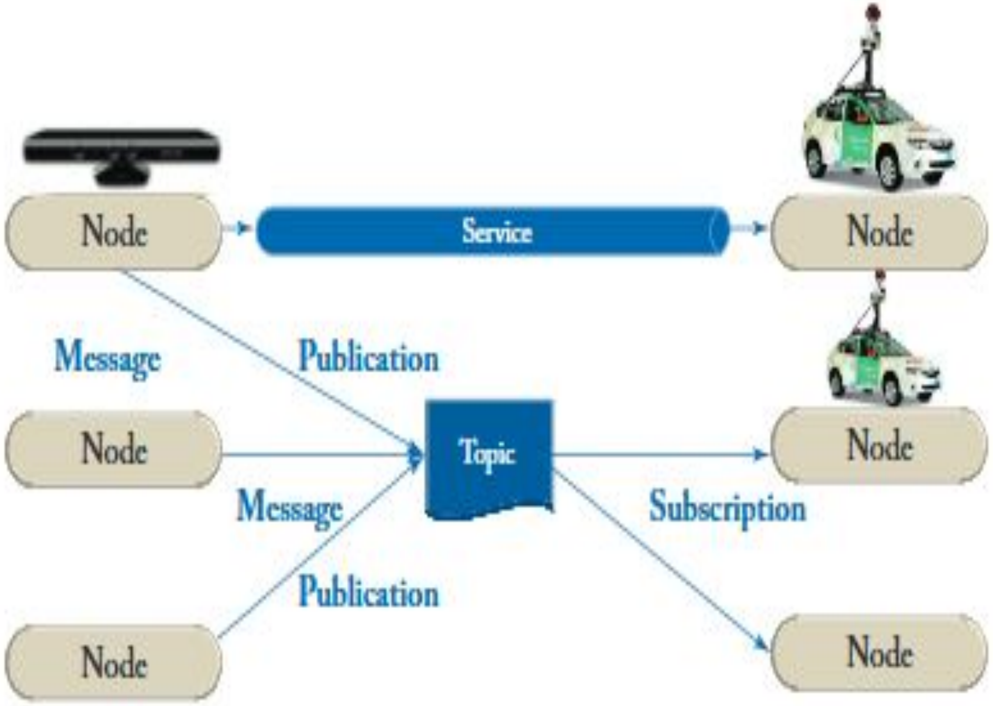
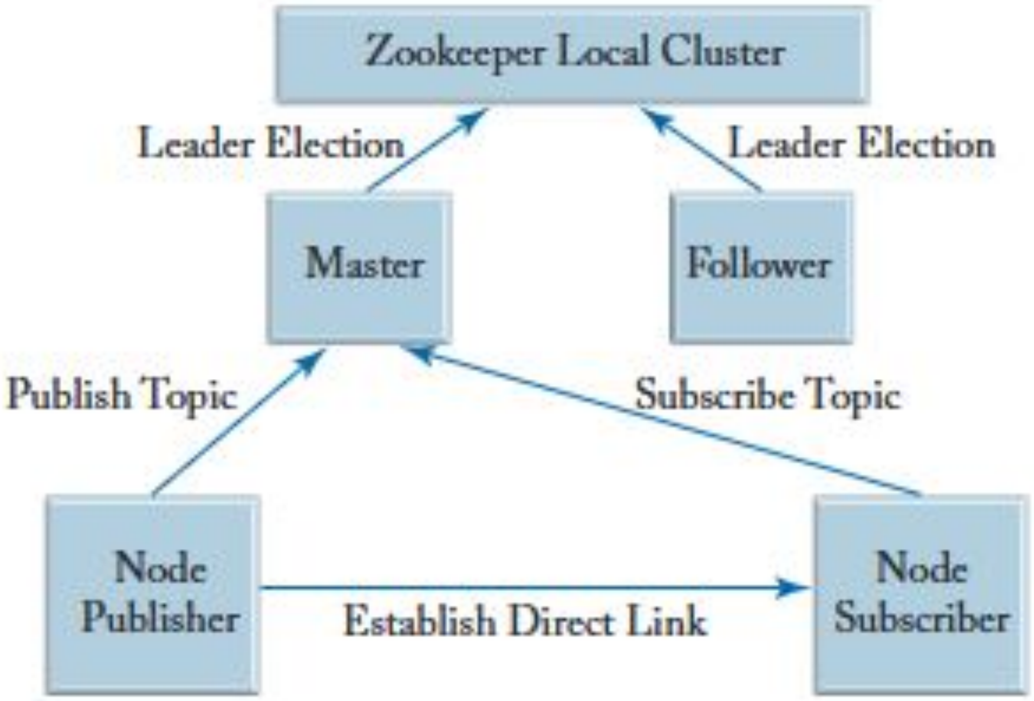


Figure 8.4: ROS communication mechanisms.



Client Systems for Autonomous Driving

Go, change the world®

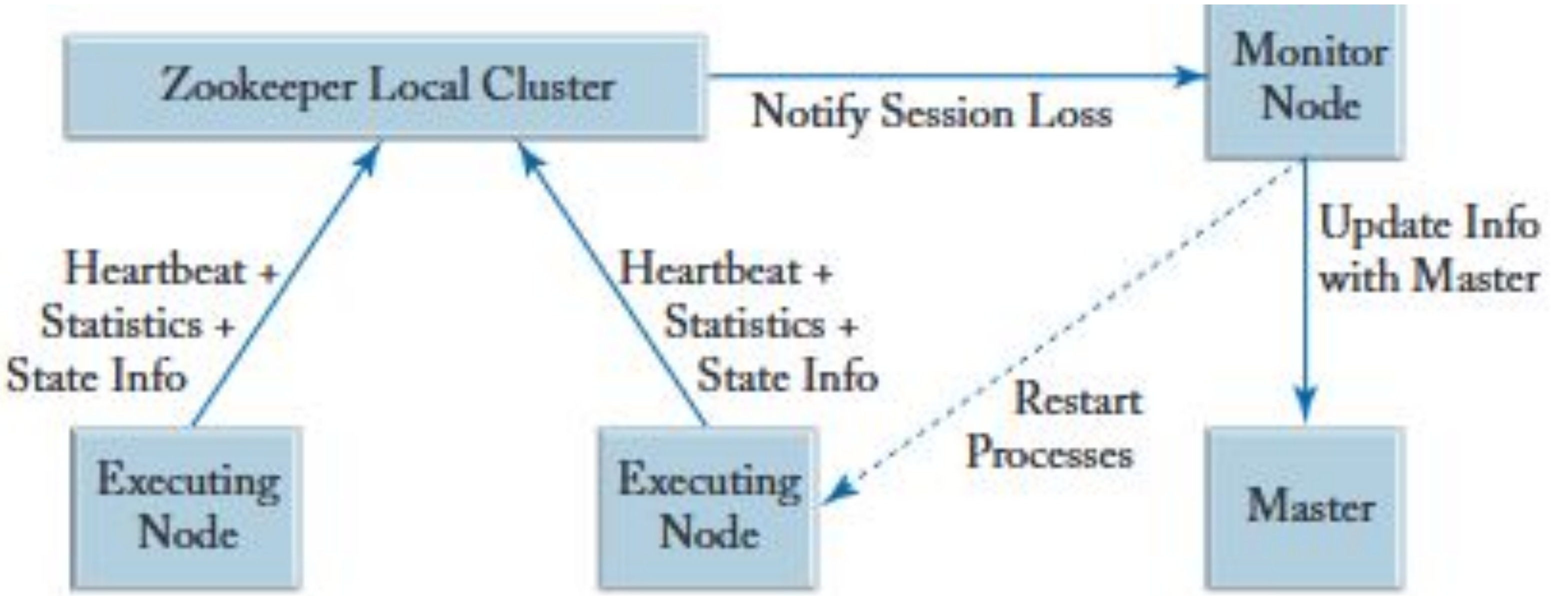


Figure 8.6: Monitor node.

- system reliability: If an autonomous vehicle is moving on the road, suddenly the ROS master node crashes, leading to a system shutdown.

- one master in the whole system, but it is not acceptable in autonomous vehicle applications.

- Thus we need to decentralize the master and achieve robustness and reliability - Zookeeper

Client Systems for Autonomous Driving - Performance

- Un-necessary delay
- Memory consumption
- Waste of computing and memory resources – due to broadcast

Existing Autonomous Driving Computing Platform

•**Hardware Configuration:**

- Two compute boxes with Intel Xeon E5 processors
- 4-8 Nvidia K80 GPUs per box
- Connected via PCI-E bus

•**Performance Metrics:**

- 12-core CPU delivers 400 GOPS/s
- Each GPU: 8 TOPS/s, 300 W power consumption
- Overall: 64.5 TOPS/s, ~3,000 W total power

•**Challenges:**

- High power consumption (~5,000 W max)
- High cost ($20,000–$30,000 per box)

•**Existing Computing Solutions**

•Overview of major computing architectures:

- GPU-based solutions
- DSP-based solutions
- FPGA-based solutions
- ASIC-based solutions

GPU-Based Computing Solutions

- **Nvidia PX Platform**
  - 2 Tegra SoCs + 2 Pascal GPUs
  - Dedicated memory & specialized DNN instructions
  - PCI-E Gen 2 × 4 bus (4.0 GB/s)
  - Gigabit Ethernet interconnection (70 GB/s)
  - 24 TOPS/s performance
  - 2,800 images/s on AlexNet

DSP-Based Solutions

- **Texas Instruments TDA2x SoC**
  - 2 floating-point C66x DSP cores
  - 4 Vision Accelerators (8× acceleration vs. Cortex-A15 CPU)
- **CEVA XM4 DSP**
  - Optimized for computer vision tasks
  - Energy-efficient: <30 mW for 1080p at 30 FPS

Client Systems for Autonomous Driving - COMPUTING PLATFORM

Go, change the world®

FPGA-Based Solutions

- **Altera Cyclone V SoC** (Used in Audi vehicles)
  - Optimized for sensor fusion & object detection
- **Xilinx Zynq UltraScale MPSoC**
  - CNN processing: 14 images/sec/Watt
  - Object tracking: 60 FPS for 1080p live stream

ASIC-Based Solutions

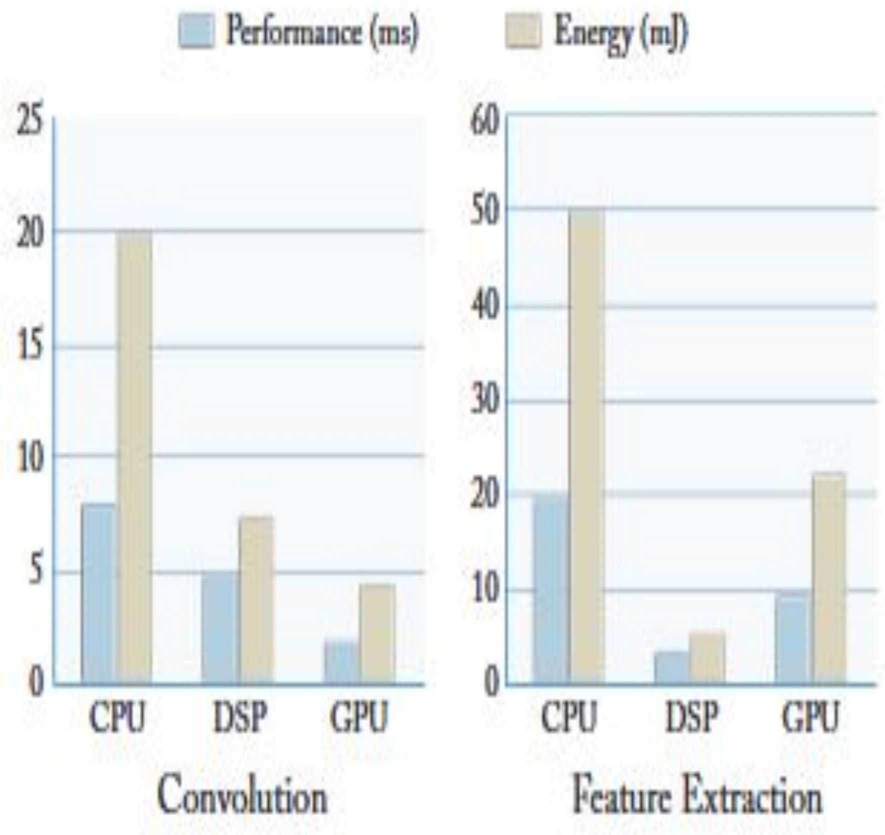
- **MobilEye EyeQ5**
  - Heterogeneous, fully programmable accelerators
  - Optimized for computer vision, signal processing, ML tasks
  - Multiple EyeQ5 devices interconnected via PCI-E

Computer Architecture Design Exploration

- Key research questions:
  - Best computing units for specific workloads
  - Feasibility of mobile processors for autonomous driving
  - Efficient platform design strategies

Matching Workloads to Computing Units

- **Convolution & Feature Extraction Workloads:**
  - CPU: 8 ms/convolution, 20 mJ
  - GPU: 2 ms/convolution, 4.5 mJ (most efficient for convolution)
  - DSP: 5 ms/convolution, 7.5 mJ
- **Feature Extraction Task:**
  - CPU: 20 ms, 50 mJ
  - GPU: 10 ms, 22.5 mJ
  - DSP: 4 ms, 6 mJ (most efficient for feature extraction)
- **Conclusion:**
  - GPUs excel at convolution tasks
  - DSPs excel at feature processing
  - CPUs handle control-heavy tasks better



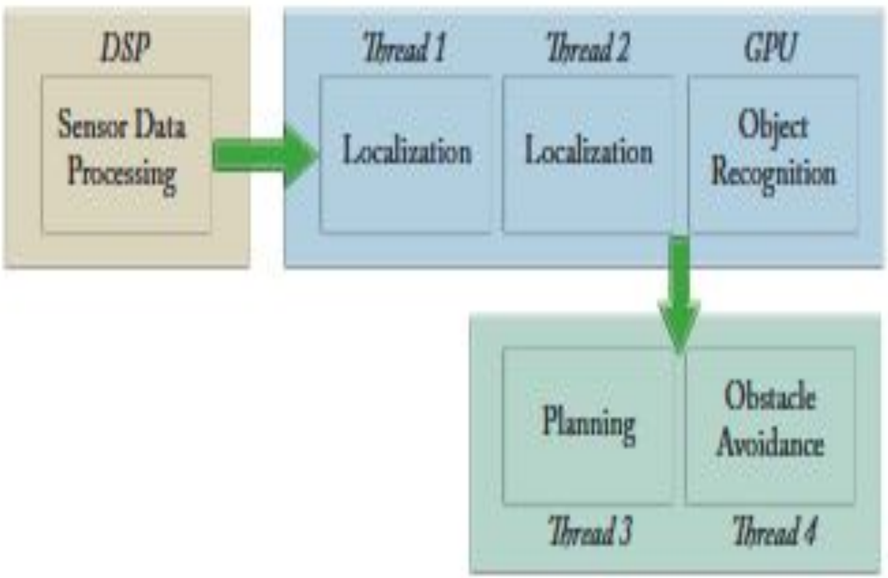


Figure 8.8: Autonomous navigation system on mobile SoC.

ARM Mobile SoC Performance

- **Localization Pipeline:**
  - Processes 25 images/sec (almost keeping up with 30 images/sec generation)
- **Deep Learning Pipeline:**
  - 2-3 object recognition tasks per second

- **Planning & Control Pipeline:**
  - Plans a path within 6 ms
- **Power Consumption:**
  - Average consumption: 11 W

- **Performance Observations:**
  - Vehicle maintained localization at 5 mph
  - With more computing resources, higher speeds could be achieved
  - Potential for a production-level autonomous driving system

Client Systems for Autonomous Driving

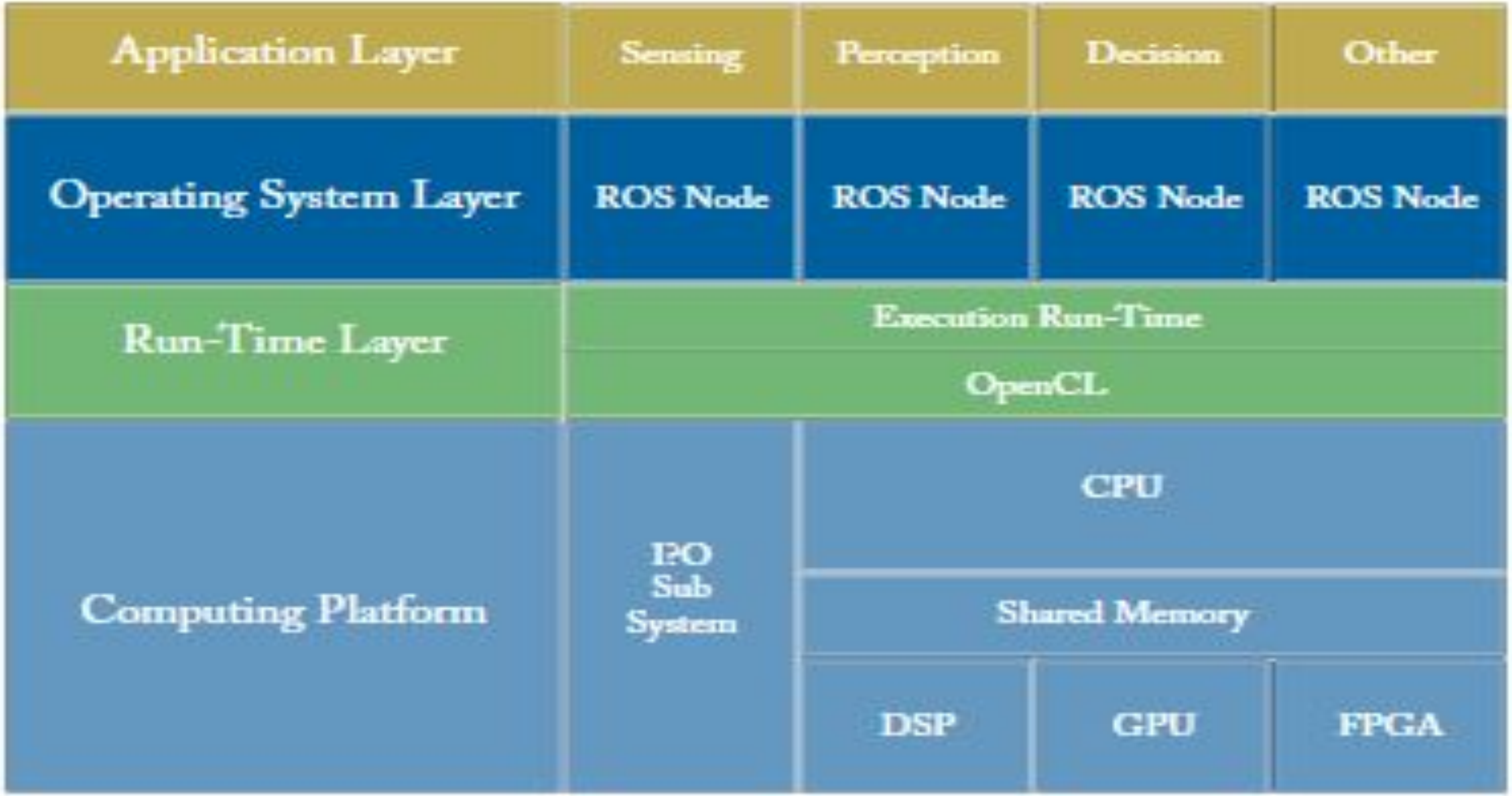


Figure 8.9: Computing stack for autonomous driving.

- **Application Layer (Top - Yellow)**
- Handles high-level tasks such as:
  - **Sensing**: Collecting data from cameras, LiDAR, radar, and other sensors.
  - **Perception**: Processing sensory data for object detection, localization, and mapping.
  - **Decision**: Making driving decisions (e.g., lane changes, braking).
  - **Other**: Additional functionalities like communication with external systems.

- **Operating System Layer (Dark Blue)**
- Composed of **ROS (Robot Operating System) Nodes**, which manage different software modules required for autonomous driving.

- Each **ROS Node** handles specific tasks like perception, decision-making, or sensor data processing.

Client Systems for Autonomous Driving

- **Run-Time Layer (Green)**
  - Includes **Execution Run-Time** and **OpenCL**.
  - **Execution Run-Time** manages the execution of tasks in real-time.
  - **OpenCL** provides a framework for running applications on heterogeneous hardware (CPU, GPU, FPGA).
- **Computing Platform (Bottom - Blue)**
- The hardware foundation for autonomous driving computations.
  - **CPU (Central Processing Unit)**: Handles general computing tasks.
  - **GPU (Graphics Processing Unit)**: Optimized for deep learning and parallel processing tasks.
  - **DSP (Digital Signal Processor)**: Specialized for low-power, high-speed signal processing.
  - **FPGA (Field-Programmable Gate Array)**: Provides flexible, energy-efficient hardware acceleration.
  - **Shared Memory**: Enables data sharing between different computing units.
  - **I/O Subsystem**: Manages data transfer between sensors and the computing platform.