ROS2 Command Logics - Material Handling System

1. ROS2 Communication Architecture

1.1 WebSocket Bridge Communication

```
// Connection establishment
var ros = new ROSLIB.Ros({ url: 'ws://localhost:9090' });
```

What happens here:

- Creates WebSocket connection to ROSBridge server running on port 9090
- ROSBridge acts as translator between web browser and ROS2 system
- Enables bidirectional communication: Web ↔ ROSBridge ↔ ROS2 Network

1.2 Communication Flow Diagram

```
Web Browser (JavaScript)
  ↓ WebSocket (JSON)
ROSBridge Server (Python/C++)
  ↓ ROS2 Native (DDS)
ROS2 Nodes (Robot Hardware)
```

2. Topic-Based Communication

2.1 Command Velocity Publisher (/cmd_vel)

Topic Setup

```
var cmdVel = new ROSLIB.Topic({
  ros: ros,                          // ROS connection object
  name: '/cmd_vel',                  // Topic name
  messageType: 'geometry_msgs/Twist'     // Message type
});
```

Message Structure

```
geometry_msgs/Twist:
├── linear (Vector3)
│   ├── x: forward/backward velocity
│   ├── y: left/right velocity (usually 0)
│   └── z: up/down velocity (usually 0)
└── angular (Vector3)
    ├── x: roll rate (usually 0)
```

```
│── y: pitch rate (usually 0)
└── z: yaw rate (turning left/right)
```

```
// geometry_msgs/Twist message format
{
  linear: {
    x: 0.5,   // Forward/backward velocity (m/s)
    y: 0.0,   // Left/right velocity (m/s) - usually 0 for differential drive
    z: 0.0    // Up/down velocity (m/s) - usually 0 for ground robots
  },
  angular: {
    x: 0.0,   // Roll angular velocity (rad/s) - usually 0
    y: 0.0,   // Pitch angular velocity (rad/s) - usually 0
    z: 1.2    // Yaw angular velocity (rad/s) - turning left/right
  }
}
```

Publishing Commands

```
function publishVelocity(linear, angular) {
  if (!cmdVel || !isConnected) return;

  // Create ROS message
  var twist = new ROSLIB.Message({
    linear: { x: linear, y: 0, z: 0 },
    angular: { x: 0, y: 0, z: angular }
  });

  // Publish to ROS2 network
  cmdVel.publish(twist);
}
```

Command Flow:

1. **JavaScript** creates Twist message with velocities
2. **ROSBridge** converts JSON to ROS2 native message
3. **ROS2 DDS** distributes message to all subscribers
4. **Robot Hardware Node** receives and executes motor commands

2.2 Odometry Subscriber (/odom)

Topic Setup

```
var odomSubscriber = new ROSLIB.Topic({
  ros: ros,
  name: '/odom',              // Standard odometry topic
```

```
  messageType: 'nav_msgs/Odometry'     // Standard odometry message
});
```

Message Structure Analysis

```
// nav_msgs/Odometry message structure
{
  header: {
    stamp: {sec: 1234567890, nanosec: 123456789},  // Timestamp
    frame_id: "odom"                          // Reference frame
  },
  child_frame_id: "base_link",                  // Robot frame
  pose: {
    pose: {
      position: {
        x: 1.23,    // Robot X position (meters)
        y: 4.56,    // Robot Y position (meters)
        z: 0.0      // Robot Z position (usually 0)
      },
      orientation: {
        x: 0.0,     // Quaternion X
        y: 0.0,     // Quaternion Y
        z: 0.707,   // Quaternion Z
        w: 0.707    // Quaternion W (represents rotation)
      }
    },
    covariance: [36 floating point values]  // Position uncertainty
  },
  twist: {
    twist: {
      linear: {
        x: 0.2,     // Current linear velocity
        y: 0.0,
        z: 0.0
      },
      angular: {
        x: 0.0,
        y: 0.0,
        z: 0.1      // Current angular velocity
      }
    },
    covariance: [36 floating point values]  // Velocity uncertainty
  }
}
```

Subscription and Data Processing

```
odomSubscriber.subscribe(function(message) {
  // Extract position
  robotPosition.x = message.pose.pose.position.x;
  robotPosition.y = message.pose.pose.position.y;

  // Convert quaternion to Euler angle (yaw)
  var orientation = message.pose.pose.orientation;
  robotOrientation = Math.atan2(
    2 * (orientation.w * orientation.z + orientation.x * orientation.y),
    1 - 2 * (orientation.y * orientation.y + orientation.z * orientation.z)
  );

  // Calculate speed from velocity
  var velocity = message.twist.twist.linear;
  var speed = Math.sqrt(velocity.x * velocity.x + velocity.y * velocity.y);

  // Update UI and control logic
  updateDisplays();
  updateNavigationControl();
});
```

3. ROS2 Command Processing Logic

3.1 Control Loop Architecture

Real-time Control Cycle (10 Hz)

// Loop for Every 100ms:

1. Read robot position from /odom
2. Calculate error (target - current)
3. Apply control algorithm (P/PID/Pure Pursuit)
4. Generate velocity commands
5. Publish to /cmd_vel
6. Repeat

```
// Navigation control timer - runs every 100ms
navigationInterval = setInterval(function() {
  if (systemState !== 'RUNNING' || !currentNavigationTarget) {
    stopNavigation();
    return;
  }

  updateNavigationControl();  // Calculate new velocities
}, 100); // 10 Hz update rate
```

Control Algorithm Processing

```
function updateNavigationControl() {
  // 1. Get current target location
  var target = locations[currentNavigationTarget];

  // 2. Calculate errors
  var dx = target.x - robotPosition.x;
  var dy = target.y - robotPosition.y;
  var distance = Math.sqrt(dx*dx + dy*dy);

  // 3. Check arrival condition
  if (distance < 0.15) {
    stopNavigation();
    return;
  }

  // 4. Apply selected control algorithm
  var velocities;
  switch(currentControlMode) {
    case 'proportional':
      velocities = calculateProportionalControl(target, distance, dx, dy);
      break;
    case 'pid':
      velocities = calculatePIDControl(target, distance, dx, dy);
      break;
    // ... other algorithms
  }

  // 5. Apply velocity smoothing
  var smoothedVel = velocitySmoother.smooth(velocities.linear, velocities.angular);

  // 6. Publish to ROS2
  publishVelocity(smoothedVel.linear, smoothedVel.angular);
}
```

3.2 Command Generation Algorithms

Proportional Control Logic

```
function calculateProportionalControl(target, distance, dx, dy) {
  var params = controlParams.proportional;

  // Calculate desired heading to target
  var targetHeading = Math.atan2(dy, dx);
```

```
  // Calculate heading error (-π to π)
  var headingError = normalizeAngle(targetHeading - robotOrientation);

  // Proportional control law
  var linearVel = Math.min(params.kp_linear * distance, params.max_linear);
  var angularVel = Math.max(Math.min(params.kp_angular * headingError,
                    params.max_angular),
                 -params.max_angular);

  return { linear: linearVel, angular: angularVel };
}
```

Mathematical Logic:

- **Linear Velocity**: v = min(Kp_linear × distance_error, v_max)
- **Angular Velocity**: ω = constrain(Kp_angular × heading_error, ±ω_max)
- **Heading Error**: θ_error = atan2(dy, dx) - θ_current

PID Control Logic

```
function calculatePIDControl(target, distance, dx, dy) {
  var params = controlParams.pid;
  var dt = pidState.dt;

  var targetHeading = Math.atan2(dy, dx);
  var headingError = normalizeAngle(targetHeading - robotOrientation);

  // Update integral terms (with windup protection)
  pidState.linearErrorIntegral += distance * dt;
  pidState.linearErrorIntegral = Math.max(Math.min(pidState.linearErrorIntegral,
                      params.integral_limit),
                     -params.integral_limit);

  pidState.angularErrorIntegral += headingError * dt;
  pidState.angularErrorIntegral = Math.max(Math.min(pidState.angularErrorIntegral,
                      params.integral_limit),
                     -params.integral_limit);

  // Calculate derivative terms
  var linearErrorDerivative = (distance - pidState.prevLinearError) / dt;
  var angularErrorDerivative = (headingError - pidState.prevAngularError) / dt;

  // PID control law
  var linearVel = params.kp_linear * distance +
         params.ki_linear * pidState.linearErrorIntegral +
         params.kd_linear * linearErrorDerivative;
```

```
  var angularVel = params.kp_angular * headingError +
          params.ki_angular * pidState.angularErrorIntegral +
          params.kd_angular * angularErrorDerivative;

  // Apply limits and store previous errors
  linearVel = Math.max(Math.min(linearVel, params.max_linear), 0);
  angularVel = Math.max(Math.min(angularVel, params.max_angular), -params.max_angular);

  pidState.prevLinearError = distance;
  pidState.prevAngularError = headingError;

  return { linear: linearVel, angular: angularVel };
}
```

Mathematical Logic:

- **PID Formula**: $u(t) = Kp \times e(t) + Ki \times \int e(t)dt + Kd \times de(t)/dt$
- **Integral Windup Protection**: Limits integral term to prevent excessive accumulation
- **Derivative Filtering**: Smooths derivative calculation to reduce noise

4. ROS2 Message Flow Analysis

4.1 Command Message Flow

```
Navigation Algorithm
  ↓ (velocities calculated)
Velocity Smoother
  ↓ (acceleration limited)
JavaScript publishVelocity()
  ↓ (ROSLIB.Message created)
ROSBridge WebSocket
  ↓ (JSON → ROS2 native)
ROS2 DDS Network
  ↓ (distributed to subscribers)
Robot Hardware Controller
  ↓ (motor commands)
Physical Robot Motors
```

4.2 Feedback Message Flow

```
Robot Wheel Encoders + IMU
  ↓ (raw sensor data)
Hardware Interface Node
```

```
    ↓ (odometry calculation)
ROS2 DDS Network (/odom topic)
    ↓ (nav_msgs/Odometry)
ROSBridge WebSocket
    ↓ (ROS2 native → JSON)
JavaScript odom callback
    ↓ (position/orientation extracted)
Navigation Algorithm Update
```
---

nav_msgs/Odometry:

```
├── header (timestamp, frame_id)
├── child_frame_id ("base_link")
├── pose
│   ├── position (x, y, z coordinates)
│   ├── orientation (quaternion: x,y,z,w)
│   └── covariance (position uncertainty)
└── twist
    ├── linear (current velocities)
    ├── angular (current rotation rates)
    └── covariance (velocity uncertainty)
```

4.3 Timing and Synchronization

Message Timestamps

```
// ROS2 messages include precise timestamps
{
  header: {
    stamp: {
      sec: 1703875234,      // Unix timestamp seconds
      nanosec: 123456789    // Nanosecond precision
    }
  }
}
```

Synchronization Logic

```
// Performance monitoring
var lastUpdateTime = Date.now();
var updateCount = 0;

odomSubscriber.subscribe(function(message) {
  // Track update frequency
  updateCount++;
  var now = Date.now();
```

```
  if (now - lastUpdateTime > 1000) {
    var frequency = updateCount; // Hz
    document.getElementById('update-rate').textContent = `${frequency}Hz`;
    updateCount = 0;
    lastUpdateTime = now;
  }

  // Process odometry data...
});
```

5. ROS2 Connection Management

5.1 Connection State Handling

```
// Connection established
ros.on('connection', function() {
  isConnected = true;
  document.getElementById('ros-status').textContent = 'CONNECTED';
  logMessage('Connected to ROS Bridge');
});

// Connection error
ros.on('error', function(error) {
  isConnected = false;
  document.getElementById('ros-status').textContent = 'ERROR';
  logMessage('ROS Connection Error: ' + error);

  // Safety: Auto-pause system on connection loss
  if (systemState === 'RUNNING') {
    pauseSystem();
    logMessage('System auto-paused due to connection loss');
  }
});

// Connection closed
ros.on('close', function() {
  isConnected = false;
  document.getElementById('ros-status').textContent = 'DISCONNECTED';
  logMessage('Disconnected from ROS Bridge');
});
```

5.2 Safety Mechanisms

```
function publishVelocity(linear, angular) {
  // Safety check: Don't send commands if disconnected
```

```
  if (!cmdVel || !isConnected) return;

  // Safety check: Emergency stop overrides
  if (emergencyStopActive) {
    linear = 0;
    angular = 0;
  }

  // Create and publish message
  var twist = new ROSLIB.Message({
    linear: { x: linear, y: 0, z: 0 },
    angular: { x: 0, y: 0, z: angular }
  });
  cmdVel.publish(twist);
}
```

----

6. Advanced ROS2 Features

6.1 Quality of Service (QoS) Configuration

```
// In real ROS2 implementation, QoS can be configured:
var cmdVel = new ROSLIB.Topic({
  ros: ros,
  name: '/cmd_vel',
  messageType: 'geometry_msgs/Twist',
  qos: {
    reliability: 'reliable',    // or 'best_effort'
    durability: 'volatile',     // or 'transient_local'
    history: 'keep_last',       // or 'keep_all'
    depth: 1                    // queue size
  }
});
```

6.2 Parameter Management

```
// Reading ROS2 parameters (if supported by ROSBridge)
var paramClient = new ROSLIB.Param({
  ros: ros,
  name: '/robot/max_velocity'
});

paramClient.get(function(value) {
  controlParams.proportional.max_linear = value;
  logMessage(`Updated max velocity: ${value}`);
});
```

6.3 Service Calls

```
// Example: Emergency stop service call
var emergencyStopService = new ROSLIB.Service({
  ros: ros,
  name: '/emergency_stop',
  serviceType: 'std_srvs/Empty'
});

function callEmergencyStop() {
  var request = new ROSLIB.ServiceRequest();

  emergencyStopService.callService(request, function(result) {
    logMessage('Emergency stop service called successfully');
  }, function(error) {
    logMessage('Emergency stop service failed: ' + error);
  });
}
```

---

7. Data Flow Optimization

7.1 Message Filtering

```
// Throttle high-frequency data to prevent UI overload
var lastOdomUpdate = 0;
var odomThrottleMs = 50; // 20 Hz max for UI updates

odomSubscriber.subscribe(function(message) {
  var now = Date.now();

  // Always update internal state
  updateRobotState(message);

  // Throttle UI updates
  if (now - lastOdomUpdate > odomThrottleMs) {
    updateUIDisplays();
    lastOdomUpdate = now;
  }

  // Always update control loop (no throttling for safety)
  updateNavigationControl();
});
```

7.2 Buffering and Interpolation

```
// Position history for smooth visualization
var positionHistory = [];
var maxHistorySize = 100;

function updateRobotState(odomMessage) {
  // Add current position to history
  positionHistory.push({
    x: odomMessage.pose.pose.position.x,
    y: odomMessage.pose.pose.position.y,
    timestamp: Date.now()
  });

  // Maintain buffer size
  if (positionHistory.length > maxHistorySize) {
    positionHistory.shift();
  }

  // Update current position
  robotPosition.x = odomMessage.pose.pose.position.x;
  robotPosition.y = odomMessage.pose.pose.position.y;
}
```

8. Error Handling and Recovery

8.1 Message Validation

```
function validateOdometryMessage(message) {
  // Check for required fields
  if (!message.pose || !message.pose.pose || !message.pose.pose.position) {
    logMessage('Invalid odometry message: missing position data');
    return false;
  }

  // Check for reasonable values
  var pos = message.pose.pose.position;
  if (Math.abs(pos.x) > 100 || Math.abs(pos.y) > 100) {
    logMessage('Warning: Robot position seems unrealistic');
    return false;
  }

  return true;
}

odomSubscriber.subscribe(function(message) {
```

```
  if (!validateOdometryMessage(message)) {
    return; // Skip invalid messages
  }

  // Process valid message...
});
```

8.2 Connection Recovery

```
var reconnectAttempts = 0;
var maxReconnectAttempts = 5;
var reconnectDelay = 5000; // 5 seconds

ros.on('close', function() {
  if (reconnectAttempts < maxReconnectAttempts) {
    reconnectAttempts++;
    logMessage(`Connection lost. Attempting reconnection ${reconnectAttempts}/${maxReconnectAttempts}...`);

    setTimeout(function() {
      try {
        ros.connect(ros.socket.url);
      } catch (error) {
        logMessage(`Reconnection attempt ${reconnectAttempts} failed: ${error}`);
      }
    }, reconnectDelay);
  } else {
    logMessage('Maximum reconnection attempts reached. Manual intervention required.');
  }
});

ros.on('connection', function() {
  reconnectAttempts = 0; // Reset counter on successful connection
});
```

This is how ROS2 commands flow through the system, from high-level navigation decisions to low-level motor control, with robust error handling and real-time performance considerations.