Robot System Architecture

Technical Documentation

Version 1.0

Context-Based Robotics Framework

June 15, 2025

Contents

1 Introduction

The Robot System Architecture implements a modern, context-based approach to robot control and management. Following React best practices and inspired by ROS principles, the system provides a clean separation of concerns with each context managing a specific domain while maintaining clear communication patterns.

1.1 Key Features

- Unified RobotContext as single source of truth
- Context-based separation of concerns
- EventBus for high-frequency updates (60Hz+)
- React state for UI synchronization
- Dynamic robot discovery and loading
- Workspace persistence
- TCP tool management
- Trajectory recording and playback
- Multiple IK solver support
- Extensible architecture

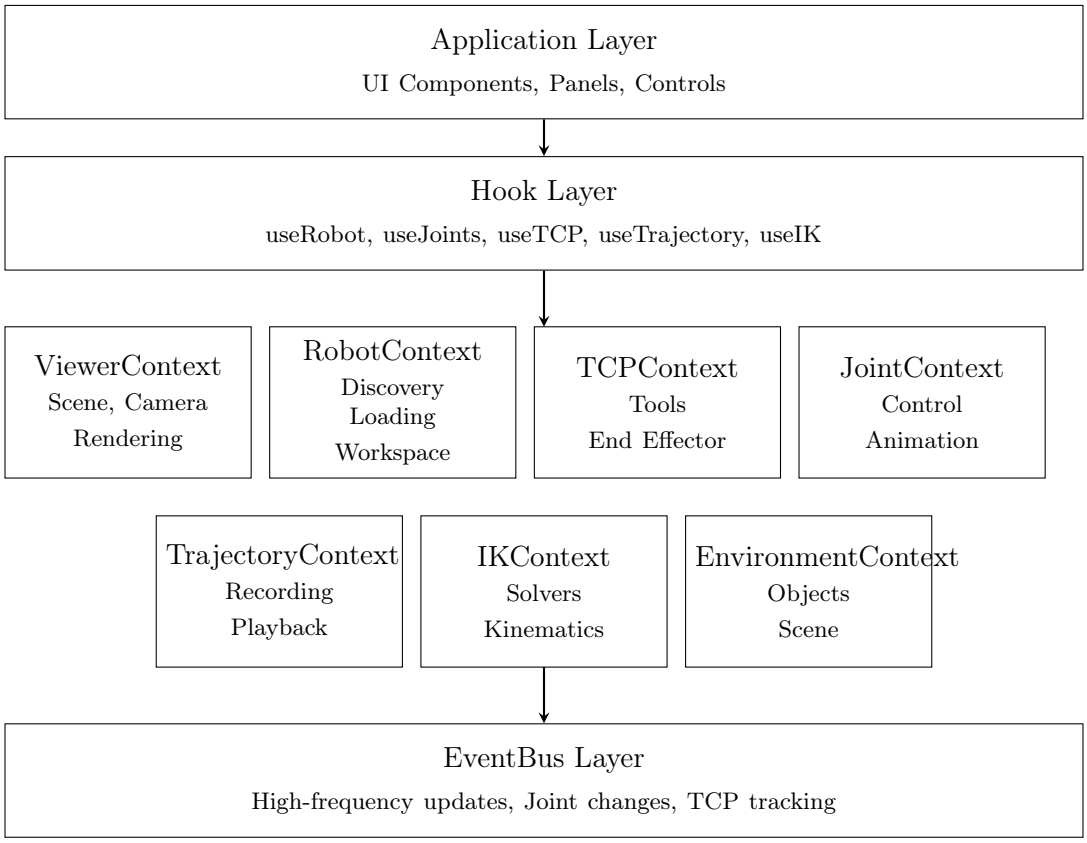
1.2 Design Principles

1. **Single Source of Truth**: Each context owns its domain data
2. **Performance First**: EventBus for real-time, state for UI
3. **Clean Interfaces**: Custom hooks abstract complexity
4. **Modular Design**: Easy to add new features via contexts
5. **Type Safety Ready**: Structure supports TypeScript migration
6. **Developer Experience**: Intuitive APIs and clear patterns

2 Architecture Overview

2.1 System Architecture

The Robot System implements a layered architecture with clear boundaries:



2.2 Data Flow Patterns

The system uses two primary data flow patterns:

Pattern	Use Case	Frequency
EventBus	Joint updates, TCP tracking, Trajectory playback	10-60Hz
React State	UI updates, Loading states, User selections	On change

Table 1: Data Flow Patterns

3 Core Components

3.1 RobotContext - The Foundation

The unified RobotContext serves as the foundation, managing all robot-related state:

3.1.1 Responsibilities

- Robot discovery from server
- Workspace management (persistence)
- URDF loading and scene integration
- Active robot tracking

- Joint control delegation
- TCP tool discovery
- Error and loading states

3.1.2 Key State Properties

Property	Description
`availableRobots`	List of discovered robots from server
`workspaceRobots`	User's selected robots (persisted)
`loadedRobots`	Map of loaded 3D robot models
`activeRobotId`	Currently selected robot ID
`activeRobot`	Currently selected robot model
`availableTools`	Discovered TCP tools

Table 2: RobotContext State Properties

3.1.3 Key Methods

Method	Description
`discoverRobots()`	Fetch available robots from server
`addRobotToWorkspace()`	Add robot to user workspace
`loadRobot()`	Load URDF and add to 3D scene
`setActiveRobotId()`	Set the active robot for control
`setJointValue()`	Control individual joint
`getJointValues()`	Get current joint configuration

Table 3: RobotContext Core Methods

3.2 Context Hierarchy

The contexts work together in a hierarchical manner:

1. **ViewerContext**: Manages 3D scene foundation
2. **RobotContext**: Central hub for robot state
3. **TCPContext**: Extends robots with tools
4. **JointContext**: Manages joint animations
5. **TrajectoryContext**: Records/plays movements
6. **IKContext**: Calculates joint configurations
7. **EnvironmentContext**: Manages scene objects

4 Creating a New Context

4.1 Step-by-Step Guide

Creating a new context follows a standard pattern. Let's create a `CollisionContext` as an example:

4.1.1 Step 1: Create Context File

Create `src/contexts/CollisionContext.jsx`:

```
import React, { createContext, useContext, useState, useCallback,
    useEffect } from 'react';
import { useRobotContext } from './RobotContext';
import { useEnvironmentContext } from './EnvironmentContext';
import EventBus from '../utils/EventBus';

const CollisionContext = createContext(null);

export const CollisionProvider = ({ children }) => {
  const { activeRobotId, getRobot } = useRobotContext();
  const { objects } = useEnvironmentContext();

  // State
  const [collisionDetection, setCollisionDetection] = useState(true);
  const [collisions, setCollisions] = useState(new Map());
  const [collisionPairs, setCollisionPairs] = useState([]);

  // Check for collisions
  const checkCollisions = useCallback(() => {
    if (!activeRobotId || !collisionDetection) return;

    const robot = getRobot(activeRobotId);
    if (!robot) return;

    const detectedCollisions = new Map();

    // Collision detection logic here
    objects.forEach(object => {
      // Simple bounding box check example
      if (checkBoundingBoxCollision(robot, object)) {
        detectedCollisions.set(object.id, {
          robotId: activeRobotId,
          objectId: object.id,
          type: 'bounding-box'
        });
      }
    });

    setCollisions(detectedCollisions);

    // Emit event if collisions changed
    if (detectedCollisions.size !== collisions.size) {
      EventBus.emit('collision:detected', {
        robotId: activeRobotId,
```

```
        collisions: Array.from(detectedCollisions.values())
      });
    }
  }, [activeRobotId, objects, collisionDetection, getRobot,
      collisions]);

  // Listen for joint changes to recheck collisions
  useEffect(() => {
    const handleJointChange = (data) => {
      if (data.robotId === activeRobotId) {
        checkCollisions();
      }
    };

    const unsubscribe = EventBus.on('robot:joint-changed',
       handleJointChange);
    return () => unsubscribe();
  }, [activeRobotId, checkCollisions]);

  // Public API
  const value = {
    // State
    collisionDetection,
    collisions,
    hasCollisions: collisions.size > 0,

    // Methods
    setCollisionDetection,
    checkCollisions,
    clearCollisions: () => setCollisions(new Map()),

    // Getters
    getCollisionsForRobot: (robotId) => {
      return Array.from(collisions.values())
        .filter(c => c.robotId === robotId);
    }
  };

  return (
    <CollisionContext.Provider value={value}>
      {children}
    </CollisionContext.Provider>
  );
};

export const useCollisionContext = () => {
  const context = useContext(CollisionContext);
  if (!context) {
    throw new Error('useCollisionContext must be used within
       CollisionProvider');
  }
  return context;
};

export default CollisionContext;
```

Listing 1: CollisionContext.jsx

4.1.2 Step 2: Create Custom Hook

Create `src/contexts/hooks/useCollision.js`:

```
import { useCallback, useEffect, useState } from 'react';
import { useCollisionContext } from '../CollisionContext';
import { useRobotSelection } from './useRobot';

export const useCollision = (robotId = null) => {
  const context = useCollisionContext();
  const { activeId: activeRobotId } = useRobotSelection();

  // Use provided robotId or fall back to active robot
  const targetRobotId = robotId || activeRobotId;

  // Local state for UI
  const [isColliding, setIsColliding] = useState(false);

  // Check collision state
  useEffect(() => {
    const collisions = context.getCollisionsForRobot(targetRobotId);
    setIsColliding(collisions.length > 0);
  }, [context, targetRobotId]);

  // Methods
  const enableDetection = useCallback(() => {
    context.setCollisionDetection(true);
  }, [context]);

  const disableDetection = useCallback(() => {
    context.setCollisionDetection(false);
  }, [context]);

  return {
    // State
    robotId: targetRobotId,
    isColliding,
    collisions: context.getCollisionsForRobot(targetRobotId),
    detectionEnabled: context.collisionDetection,

    // Methods
    enableDetection,
    disableDetection,
    checkNow: context.checkCollisions,

    // Convenience
    hasCollisions: isColliding,
    collisionCount: context.getCollisionsForRobot(targetRobotId).
        length
  };
};

```

```
// Specialized hook for collision monitoring
export const useCollisionMonitor = () => {
  const { collisions, hasCollisions } = useCollisionContext();
  const [collisionHistory, setCollisionHistory] = useState([]);

  useEffect(() => {
    if (hasCollisions) {
      setCollisionHistory(prev => [...prev, {
        timestamp: Date.now(),
        count: collisions.size,
        collisions: Array.from(collisions.values())
      }]);
    }
  }, [collisions, hasCollisions]);

  return {
    currentCollisions: Array.from(collisions.values()),
    hasCollisions,
    history: collisionHistory,
    clearHistory: () => setCollisionHistory([])
  };
};

export default useCollision;
```

Listing 2: useCollision.js

4.1.3 Step 3: Add to Provider Chain

Update `src/App.jsx`:

```
import { CollisionProvider } from './contexts/CollisionContext';

const App = () => {
  return (
    <ViewerProvider>
      <RobotProvider>
        <EnvironmentProvider>
          <TCPProvider>
            <JointProvider>
              <TrajectoryProvider>
                <IKProvider>
                  <CollisionProvider>  {/* Add here */}
                    <WorldProvider>
                      <AppContent />
                    </WorldProvider>
                  </CollisionProvider>
                </IKProvider>
              </TrajectoryProvider>
            </JointProvider>
          </TCPProvider>
        </EnvironmentProvider>
      </RobotProvider>
    </ViewerProvider>
  );
};
```

Listing 3: App.jsx Provider Chain

4.1.4 Step 4: Create UI Component

Create a component that uses the new context:

```jsx
import React from 'react';
import { useCollision } from '../../contexts/hooks/useCollision';

const CollisionIndicator = () => {
  const {
    isColliding,
    collisionCount,
    detectionEnabled,
    enableDetection,
    disableDetection
  } = useCollision();

  return (
    <div className={`collision-indicator ${isColliding ? 'danger' : 'safe'}`}>
      <div className="status">
        <span className="icon">
          {isColliding ? '      ' : '   '}
        </span>
        <span className="text">
          {isColliding
            ? `${collisionCount} collision(s) detected!`
            : 'No collisions'}
        </span>
      </div>

      <button
        onClick={detectionEnabled ? disableDetection : enableDetection}
        className={`toggle-btn ${detectionEnabled ? 'active' : ''}`}
      >
        {detectionEnabled ? 'Disable' : 'Enable'} Detection
      </button>
    </div>
  );
};

export default CollisionIndicator;
```

Listing 4: CollisionIndicator.jsx

5 Context Design Patterns

5.1 State Management Patterns

5.1.1 Pattern 1: UI State in Context

Use React state for UI-related data:

```
const [isLoading, setIsLoading] = useState(false);
const [error, setError] = useState(null);
const [selectedItems, setSelectedItems] = useState([]);
```

Listing 5: UI State Pattern

5.1.2 Pattern 2: High-Frequency State via EventBus

Use EventBus for data that changes frequently:

```
// Emitter
EventBus.emit('robot:joint-changed', {
  robotId,
  jointName,
  value,
  timestamp: Date.now()
});

// Listener
useEffect(() => {
  const handleJointChange = (data) => {
    // React to changes
  };

  const unsubscribe = EventBus.on('robot:joint-changed',
     handleJointChange);
  return () => unsubscribe();
}, []);
```

Listing 6: EventBus Pattern

5.1.3 Pattern 3: Computed Properties

Provide computed properties for convenience:

```
const value = {
  // Raw state
  robots,

  // Computed properties
  hasRobots: robots.size > 0,
  robotCount: robots.size,
  robotNames: Array.from(robots.keys()),
  isEmpty: robots.size === 0
};
```

Listing 7: Computed Properties Pattern

5.2 Hook Design Patterns

5.2.1 Pattern 1: Default Parameter Hook

Accept optional robotId, default to active:

```
export const useFeature = (robotId = null) => {
  const { activeId } = useRobotSelection();
  const targetRobotId = robotId || activeId;

  // Use targetRobotId throughout
};
```

Listing 8: Default Parameter Pattern

5.2.2 Pattern 2: Specialized Hooks

Create focused hooks for specific use cases:

```
// General hook
export const useRobot = () => { /* ... */ };

// Specialized hooks
export const useRobotWorkspace = () => { /* ... */ };
export const useRobotDiscovery = () => { /* ... */ };
export const useRobotLoading = () => { /* ... */ };
```

Listing 9: Specialized Hook Pattern

5.2.3 Pattern 3: State and Methods Grouping

Group related state and methods:

```
return {
  // State group
  state: {
    robots,
    isLoading,
    error
  },

  // Action group
  actions: {
    load: loadRobot,
    remove: removeRobot,
    refresh: discoverRobots
  },

  // Getters group
  getters: {
    getRobot,
    getRobotCount,
    hasRobots
  }
};
```

Listing 10: Grouping Pattern

6 Performance Optimization

6.1 When to Use EventBus vs State

Criteria	Use EventBus	Use React State
Update Frequency	>10Hz	<10Hz
Consumers	Multiple contexts	Single context
Data Type	Transient updates	Persistent state
Example	Joint positions	Loading state

Table 4: EventBus vs React State Decision Matrix

6.2 Optimization Techniques

6.2.1 Memoization

Use React memoization for expensive computations:

```
const expensiveComputation = useMemo(() => {
  return robots.filter(robot => {
    // Complex filtering logic
    return robot.type === 'industrial' && robot.dof >= 6;
  });
}, [robots]);

const memoizedCallback = useCallback((robotId) => {
  return robots.get(robotId);
}, [robots]);
```

Listing 11: Memoization Example

6.2.2 Batching Updates

Batch related state updates:

```
// Instead of multiple setState calls
setRobots(newRobots);
setActiveRobotId(robotId);
setIsLoading(false);

// Use a single state update
setState(prev => ({
  ...prev,
  robots: newRobots,
  activeRobotId: robotId,
  isLoading: false
}));
```

Listing 12: Batching Updates

7 Testing Contexts

7.1 Unit Testing

Test contexts in isolation:

```
import { renderHook, act } from '@testing-library/react-hooks';
import { RobotProvider, useRobotContext } from '../RobotContext';

describe('RobotContext', () => {
  const wrapper = ({ children }) => (
    <RobotProvider>{children}</RobotProvider>
  );

  test('should load robot', async () => {
    const { result } = renderHook(() => useRobotContext(), { wrapper
        });

    await act(async () => {
      await result.current.loadRobot('test-robot', '/path/to/urdf');
    });

    expect(result.current.loadedRobots.has('test-robot')).toBe(true);
  });

  test('should set active robot', () => {
    const { result } = renderHook(() => useRobotContext(), { wrapper
        });

    act(() => {
      result.current.setActiveRobotId('test-robot');
    });

    expect(result.current.activeRobotId).toBe('test-robot');
  });
});
```

Listing 13: Context Unit Test

7.2 Integration Testing

Test context interactions:

```
describe('Robot System Integration', () => {
  test('TCP updates when robot joints change', async () => {
    const { result: robotResult } = renderHook(() => useRobot());
    const { result: tcpResult } = renderHook(() => useTCP());

    // Load robot
    await act(async () => {
      await robotResult.current.loadRobot('ur5', '/ur5.urdf');
    });

    // Change joint
    act(() => {
```

```
      robotResult.current.setJointValue('ur5', 'joint1', 0.5);
    });

    // Verify TCP updated
    expect(tcpResult.current.currentEndEffectorPoint).not.toEqual({
      x: 0, y: 0, z: 0
    });
  });
});
```

Listing 14: Integration Test

8 Best Practices

8.1 Context Design

1. **Single Responsibility**: Each context manages one domain
2. **Clear Dependencies**: Document which contexts depend on others
3. **Avoid Circular Dependencies**: Use EventBus for cross-context communication
4. **Provide Defaults**: Always provide sensible default values
5. **Error Boundaries**: Wrap providers with error boundaries
6. **Type Safety**: Design with TypeScript migration in mind

8.2 Hook Design

1. **Consistent Naming**: Use `use` prefix for all hooks
2. **Return Objects**: Return objects, not arrays, for clarity
3. **Provide Conveniences**: Include computed properties
4. **Handle Loading States**: Always expose loading/error states
5. **Memoize Callbacks**: Use useCallback for stable references
6. **Document Parameters**: Clear JSDoc comments

8.3 Performance Guidelines

1. **EventBus for Real-time**: Use for >10Hz updates
2. **State for UI**: Use React state for UI updates
3. **Lazy Loading**: Load contexts only when needed
4. **Cleanup Listeners**: Always unsubscribe in useEffect cleanup
5. **Batch Updates**: Group related state changes
6. **Profile Performance**: Use React DevTools Profiler

9 Common Patterns

9.1 Robot-Specific Context

Many contexts need robot-specific data:

```
export const useRobotFeature = (robotId = null) => {
  const context = useFeatureContext();
  const { activeId } = useRobotSelection();

  const targetRobotId = robotId || activeId;

  // Get robot-specific data
  const robotData = context.getRobotData(targetRobotId);

  // Robot-specific methods
  const updateFeature = useCallback((data) => {
    return context.updateFeature(targetRobotId, data);
  }, [context, targetRobotId]);

  return {
    robotId: targetRobotId,
    ...robotData,
    updateFeature
  };
};
```

Listing 15: Robot-Specific Context Pattern

9.2 Cross-Context Communication

Use EventBus for loose coupling:

```
// In ContextA
const handleUpdate = () => {
  // Do internal work
  updateInternalState();

  // Notify other contexts
  EventBus.emit('contextA:updated', {
    robotId,
    data: relevantData
  });
};

// In ContextB
useEffect(() => {
  const handleContextAUpdate = (event) => {
    if (event.robotId === targetRobotId) {
      // React to ContextA changes
      updateBasedOnContextA(event.data);
    }
  };

  const unsubscribe = EventBus.on('contextA:updated',
     handleContextAUpdate);
```

```
23    return () => unsubscribe();
24  }, [targetRobotId]);
```

Listing 16: Cross-Context Communication

10 Migration Guide

10.1 Adding TypeScript

The architecture is designed for easy TypeScript migration:

```
1  interface RobotContextValue {
2    // State
3    robots: Map<string, Robot>;
4    activeRobotId: string | null;
5    isLoading: boolean;
6    error: string | null;
7
8    // Methods
9    loadRobot: (id: string, path: string) => Promise<Robot>;
10   setActiveRobotId: (id: string | null) => void;
11   // ... more methods
12 }
13
14 const RobotContext = createContext<RobotContextValue | null>(null);
```

Listing 17: TypeScript Context

10.2 Future Enhancements

The architecture supports future enhancements:

1. **WebSocket Integration**: Real-time robot control
2. **Multi-Robot Coordination**: Collision avoidance
3. **Cloud Persistence**: Save/load workspaces
4. **Collaborative Features**: Multi-user support
5. **Plugin System**: Dynamic feature loading
6. **Performance Monitoring**: Built-in metrics

11 Troubleshooting

11.1 Common Issues

Issue	Solution
Context not updating	Check if component is wrapped in provider
Infinite re-renders	Check useEffect dependencies
Memory leaks	Ensure EventBus listeners are cleaned up
Stale closures	Use refs for values in callbacks
Performance issues	Profile with React DevTools

Table 5: Common Issues and Solutions

11.2 Debugging Tools

```
// Add to any context for debugging
useEffect(() => {
  if (process.env.NODE_ENV === 'development') {
    console.log('[ContextName] State updated:', {
      robots: robots.size,
      activeRobotId,
      isLoading
    });
  }
}, [robots, activeRobotId, isLoading]);

// Global debug helper
window.debugContext = (contextName) => {
  EventBus.emit('debug:dump-context', { contextName });
};
```

Listing 18: Context Debugger