TMC & 511 System Integration Guide

National Event Feeds & Auto-Notification System

DOT Corridor Communicator Integration Architecture

Table of Contents

Overview

Purpose

This document describes how to integrate the DOT Corridor Communicator with Traffic Management Centers (TMCs) and 511 traveler information systems to:

- **Aggregate national event feeds** from multiple sources
- **Distribute corridor-specific messages** to TMCs and 511 systems
- **Trigger auto-notifications** for impacted corridor traffic
- **Enable bi-directional data exchange** between systems
- **Provide real-time alerts** to travelers and fleet operators

Benefits

✅ **Real-time Traffic Intelligence** - Aggregated view of corridor conditions across states ✅ **Automated Alerting** - Instant notifications when incidents affect your corridors ✅ **Multi-State Coordination** - Share events seamlessly between neighboring states ✅ **Traveler Information** - Push critical updates to 511 systems automatically ✅ **Fleet Management** - Enable commercial vehicle operators to receive corridor alerts ✅ **Grant Competitiveness** - Demonstrate advanced traffic management capabilities

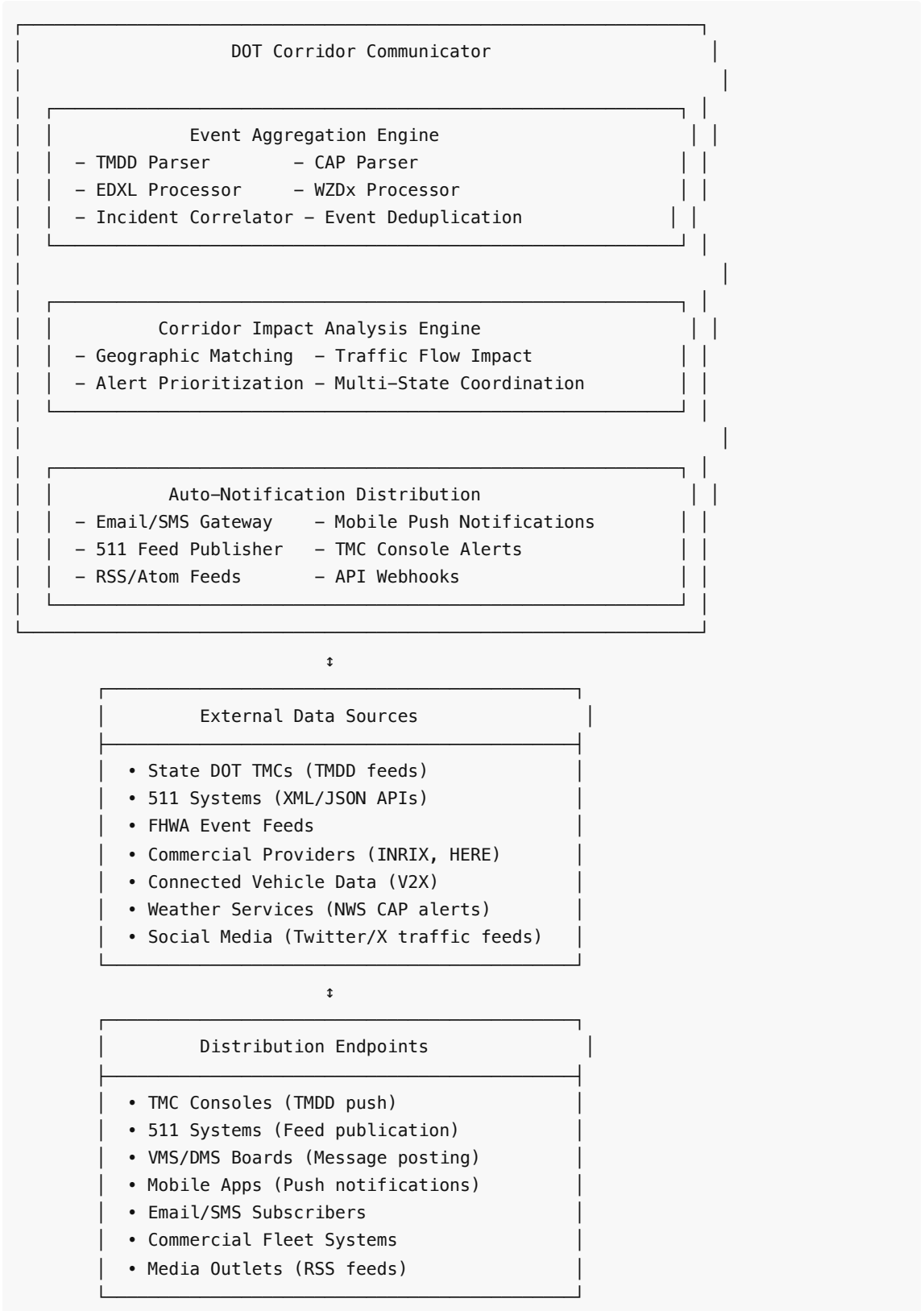
Key Standards

- **TMDD** (Traffic Management Data Dictionary) - Event data exchange
- **CAP** (Common Alerting Protocol) - Emergency alerts
- **EDXL-DE** (Emergency Data Exchange Language) - Event distribution
- **SAE J2354** - Advanced Traveler Information Systems (ATIS)
- **NTCIP 1218** - Transportation sensor systems
- **WZDx** (Work Zone Data Exchange) - Construction information

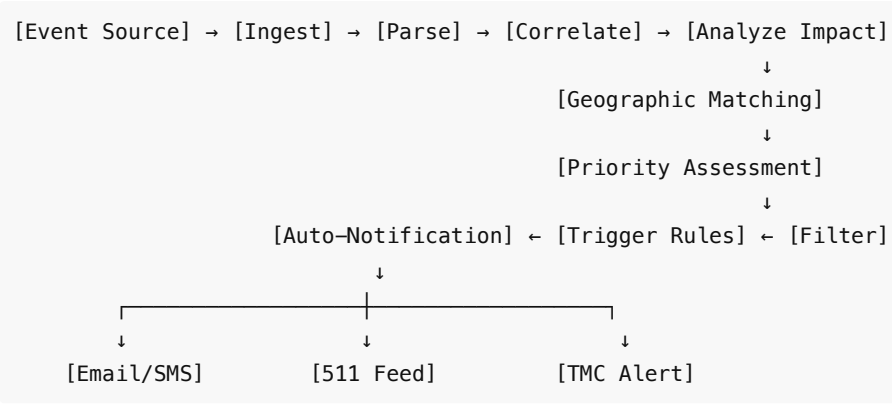
---

System Architecture

High-Level Architecture



Data Flow



Integration Standards

1. TMDD (Traffic Management Data Dictionary)

Version: TMDD v3.1 (current standard) **Format**: XML **Use**: Incident/event data exchange between TMCs

Key Message Types:

Event Reporting:

```xml
<tmdd:eventReport>
  <event-id>IA-I80-2025-001</event-id>
  <event-type>crash</event-type>
  <severity>major</severity>
  <location>
    <link-id>I-80</link-id>
    <direction>eastbound</direction>
    <milepost>100.5</milepost>
  </location>
  <impact>
    <lanes-blocked>2</lanes-blocked>
    <expected-duration>90</expected-duration>
    <queue-length>3.2</queue-length>
  </impact>
  <timestamp>2025-12-27T10:15:00Z</timestamp>
</tmdd:eventReport>
```

Equipment Status:

```xml
<tmdd:equipmentStatus>
  <equipment-id>VMS-I80-MM100</equipment-id>
  <equipment-type>dynamic-message-sign</equipment-type>
  <status>operational</status>
  <current-message>CRASH AHEAD - RIGHT LANES BLOCKED</current-message>
</tmdd:equipmentStatus>
```

Benefits:

- ✅ Standard used by all major TMC vendors (Delcan, TransSuite, ATMS.now)
- ✅ Comprehensive event taxonomy
- ✅ Built-in geographic referencing
- ✅ Impact assessment fields

2. CAP (Common Alerting Protocol)

Version: CAP v1.2 (OASIS standard) **Format**: XML **Use**: Emergency alerts, weather warnings, public safety

CAP Alert Structure:

```
<alert xmlns="urn:oasis:names:tc:emergency:cap:1.2">
  <identifier>IA-DOT-I80-ALERT-2025-001</identifier>
  <sender>iowa.dot@iowa.gov</sender>
  <sent>2025-12-27T10:15:00-06:00</sent>
  <status>Actual</status>
  <msgType>Alert</msgType>
  <scope>Public</scope>

  <info>
    <category>Transport</category>
    <event>Major Traffic Incident</event>
    <urgency>Expected</urgency>
    <severity>Severe</severity>
    <certainty>Observed</certainty>
    <headline>Major Crash on I-80 EB at MM 100</headline>
    <description>
      Multi-vehicle crash blocking right 2 lanes. Expect delays of 30+ minutes.
      Alternate route: US-6 or I-35 to I-80.
    </description>
    <instruction>Use alternate routes. Reduce speed and watch for emergency
vehicles.</instruction>

    <area>
      <areaDesc>I-80 Eastbound MM 98-105, Polk County, Iowa</areaDesc>
      <polygon>41.59,-93.65 41.59,-93.55 41.57,-93.55 41.57,-93.65
41.59,-93.65</polygon>
    </area>
  </info>
</alert>
```

Benefits:

- ✅ FEMA IPAWS (Integrated Public Alert & Warning System) compatible
- ✅ Wireless Emergency Alerts (WEA) ready
- ✅ Geographic targeting (polygon/circle)
- ✅ Multi-language support

3. EDXL-DE (Emergency Data Exchange Language - Distribution Element)

Version: EDXL-DE v1.0 **Format**: XML **Use**: Routing and packaging of emergency information

EDXL Wrapper:

```xml
<EDXLDistribution xmlns="urn:oasis:names:tc:emergency:EDXL:DE:1.0">
  <distributionID>IA-DOT-DIST-2025-001</distributionID>
  <senderID>iowa.dot@iowa.gov</senderID>
  <dateTimeSent>2025-12-27T10:15:00-06:00</dateTimeSent>
  <distributionStatus>Actual</distributionStatus>
  <distributionType>Report</distributionType>

  <targetArea>
    <circle>41.59,-93.62 10</circle> <!-- 10 mile radius -->
  </targetArea>

  <contentObject>
    <contentDescription>Traffic Incident Alert</contentDescription>
    <xmlContent>
      <!-- CAP or TMDD message here -->
    </xmlContent>
  </contentObject>
</EDXLDistribution>
```

Benefits:

- ✅ Message routing to specific recipients
- ✅ Multi-message bundling
- ✅ Digital signatures for authenticity
- ✅ Distribution tracking

4. WZDx (Work Zone Data Exchange)

Version: WZDx v4.2 **Format**: GeoJSON **Use**: Construction/maintenance zone information

WZDx Feed:

```json
{
  "feed_info": {
    "publisher": "Iowa DOT",
    "version": "4.2",
    "update_frequency": 300,
    "update_date": "2025-12-27T10:15:00Z"
  },
  "type": "FeatureCollection",
  "features": [{
    "type": "Feature",
    "properties": {
      "core_details": {
        "event_type": "work-zone",
        "road_names": ["Interstate 80"],
        "direction": "eastbound",
        "beginning_milepost": 100.0,
        "ending_milepost": 105.0,
        "start_date": "2025-06-01T06:00:00Z",
```

```
        "end_date": "2025-09-30T18:00:00Z"
      },
      "restrictions": [{
        "type": "reduced-width",
        "value": 11,
        "unit": "feet"
      }],
      "types_of_work": [{
        "type_name": "surface-work",
        "is_architectural_change": false
      }]
    },
    "geometry": {
      "type": "LineString",
      "coordinates": [
        [-93.625, 41.586],
        [-93.600, 41.588],
        [-93.575, 41.590]
      ]
    }
  }]
}
```

Benefits:

- ✅ Standard adopted by major navigation apps (Google, Apple, Waze)
- ✅ Real-time updates
- ✅ Lane-level detail
- ✅ Equipment restrictions (height, width, weight)

TMC Integration

Architecture

```
DOT Corridor Communicator ↔ TMC System
        ↓                          ↓
   [REST API]                [TMDD Feed]
        ↓                          ↓
   [WebSocket]               [NTCIP 1218]
        ↓                          ↓
   [Message Bus]             [Database]
```

Integration Methods

Method 1: TMDD XML Feed (Pull)

TMC pulls events from Corridor Communicator

Endpoint:

```
GET /api/tmc/events/tmdd
```

Query Parameters:

- `corridor` - Filter by corridor (e.g., I-80)
- `severity` - Filter by severity (minor, major, critical)
- `event_type` - Filter by type (crash, hazard, weather, etc.)
- `since` - Events since timestamp (ISO 8601)
- `bbox` - Bounding box (minLon,minLat,maxLon,maxLat)

Response:

```
<?xml version="1.0" encoding="UTF-8"?>
<tmdd:eventFeed xmlns:tmdd="http://www.tmdd.org/303/messages">
  <feed-metadata>
    <publisher>DOT Corridor Communicator</publisher>
    <update-frequency>60</update-frequency>
    <last-updated>2025-12-27T10:15:00Z</last-updated>
  </feed-metadata>

  <events>
    <event>
      <event-id>IA-I80-2025-001</event-id>
      <event-type>crash</event-type>
      <severity>major</severity>
      <!-- Full TMDD event details -->
    </event>
  </events>
</tmdd:eventFeed>
```

Update Frequency: 30-60 seconds recommended

Method 2: Real-Time WebSocket (Push)

Corridor Communicator pushes events to TMC

Connection:

```
const tmcSocket = new WebSocket('wss://api.corridor.gov/tmc/stream');

tmcSocket.on('connect', () => {
  // Subscribe to specific corridors
  tmcSocket.send(JSON.stringify({
    action: 'subscribe',
    corridors: ['I-80', 'I-35', 'I-29'],
    event_types: ['crash', 'hazard', 'weather'],
    min_severity: 'major'
  }));
});

tmcSocket.on('event', (data) => {
  // Process incoming event
  const event = JSON.parse(data);
  displayOnTMCConsole(event);
```

```
  updateTrafficMap(event);

  if (event.severity === 'critical') {
    triggerAudibleAlert();
  }
});
```

Message Format:

```
{
  "message_type": "event_update",
  "event_id": "IA-I80-2025-001",
  "timestamp": "2025-12-27T10:15:00Z",
  "event_type": "crash",
  "severity": "major",
  "location": {
    "corridor": "I-80",
    "direction": "eastbound",
    "milepost": 100.5,
    "coordinates": {
      "latitude": 41.5868,
      "longitude": -93.6250
    }
  },
  "impact": {
    "lanes_blocked": 2,
    "total_lanes": 3,
    "queue_length_miles": 3.2,
    "expected_duration_minutes": 90,
    "delay_minutes": 35
  },
  "description": "Multi-vehicle crash involving semi-truck",
  "responders": ["Iowa State Patrol", "DOT FIRST Team"],
  "cameras": ["CAM-I80-MM100-EB"],
  "vms_boards": ["VMS-I80-MM98-EB", "VMS-I80-MM95-EB"]
}
```

Benefits:

- ✅ Instant notifications (< 1 second latency)
- ✅ Reduced server load (no polling)
- ✅ Bi-directional communication
- ✅ Connection status monitoring

Method 3: NTCIP 1218 Integration (Equipment Control)

Direct control of roadside equipment

Use Cases:

- Post messages to VMS/DMS boards
- Activate warning signs
- Control ramp meters

- Adjust traffic signals

Example - Post VMS Message:

```
// Via Corridor Communicator API
POST /api/tmc/vms/post-message

{
  "equipment_id": "VMS-I80-MM95-EB",
  "message": {
    "pages": [
      {
        "text": "CRASH AHEAD",
        "display_time": 3
      },
      {
        "text": "RIGHT LANES BLOCKED",
        "display_time": 3
      },
      {
        "text": "USE CAUTION",
        "display_time": 2
      }
    ],
    "priority": "high",
    "duration": 3600,
    "triggered_by": "event:IA-I80-2025-001"
  }
}
```

Corridor Communicator translates to NTCIP 1218:

```
SET dmsMessageMultiString.1.1.0 = "[cr3][jl3][fo3]CRASH AHEAD[cr3][jl3][fo3]RIGHT
LANES BLOCKED[cr3][jl3][fo3]USE CAUTION"
SET dmsMessagePriority.1.1.0 = 255
SET dmsActivateMessage.1.1.0 = 1
```

TMC Console Integration

Display Event on TMC Map:

```
// Corridor Communicator provides map-ready data
GET /api/tmc/events/map-layer

// Returns GeoJSON for direct display
{
  "type": "FeatureCollection",
  "features": [{
    "type": "Feature",
    "geometry": {
      "type": "Point",
      "coordinates": [-93.6250, 41.5868]
```

```
    },
    "properties": {
      "event_id": "IA-I80-2025-001",
      "event_type": "crash",
      "severity": "major",
      "icon": "crash-major",
      "popup_html": "<strong>Major Crash</strong><br>I-80 EB @ MM 100.5<br>2 lanes
blocked<br>35 min delay",
      "style": {
        "color": "#FF0000",
        "size": "large",
        "blink": true
      }
    }
  }]
}
```

---

511 System Integration

Overview

511 systems provide traveler information via:

- **Phone** (dial 511)
- **Web** (511ia.org, 511ne.org, etc.)
- **Mobile Apps** (iOS/Android)
- **Social Media** (Twitter/X, Facebook)
- **Email/SMS Alerts**

Integration Architecture

```
Corridor Communicator → 511 System
         ↓                    ↓
    [Event Feed]         [IVR System]
         ↓                    ↓
    [CAP Alerts]         [Web Display]
         ↓                    ↓
    [RSS Feeds]          [Mobile Apps]
         ↓                    ↓
   [API Webhooks]        [Alert Subscriptions]
```

Method 1: XML Event Feed

511 System polls Corridor Communicator

Endpoint:

```
GET /api/511/events/xml
```

Response (SAE J2354 compliant):

```xml
<?xml version="1.0" encoding="UTF-8"?>
<traffic-data xmlns="http://www.sae.org/j2354">
  <metadata>
    <provider>Iowa DOT Corridor Communicator</provider>
    <timestamp>2025-12-27T10:15:00Z</timestamp>
    <coverage-area>
      <state>IA</state>
      <corridors>I-80, I-35, I-29, US-30</corridors>
    </coverage-area>
  </metadata>

  <incidents>
    <incident>
      <id>IA-I80-2025-001</id>
      <type>crash</type>
      <severity>major</severity>
      <reported-time>2025-12-27T10:05:00Z</reported-time>
      <location>
        <route>I-80</route>
        <direction>eastbound</direction>
        <cross-street>Exit 125 (Altoona)</cross-street>
        <county>Polk</county>
        <latitude>41.5868</latitude>
        <longitude>-93.6250</longitude>
      </location>
      <impact>
        <affected-lanes>right 2 lanes</affected-lanes>
        <delay>35 minutes</delay>
        <queue>3.2 miles</queue>
      </impact>
      <description>Multi-vehicle crash. Emergency crews on scene.</description>
      <advice>Use alternate route: US-6 or I-35 to I-80 west of Des Moines.</advice>
    </incident>
  </incidents>

  <road-conditions>
    <segment>
      <route>I-80</route>
      <direction>both</direction>
      <from-milepost>100</from-milepost>
      <to-milepost>105</to-milepost>
      <condition>wet</condition>
      <visibility>reduced</visibility>
      <temperature>34</temperature>
      <advisory>Patchy fog. Use caution.</advisory>
    </segment>
  </road-conditions>
</traffic-data>
```

Method 2: JSON REST API

Modern 511 systems prefer JSON

Endpoint:

```
GET /api/511/events/json?format=v2
```

Response:


```
{
  "metadata": {
    "version": "2.0",
    "provider": "Iowa DOT Corridor Communicator",
    "timestamp": "2025-12-27T10:15:00Z",
    "update_frequency": 60,
    "coverage": {
      "state": "IA",
      "corridors": ["I-80", "I-35", "I-29", "US-30"]
    }
  },
  "incidents": [{
    "id": "IA-I80-2025-001",
    "type": "crash",
    "severity": "major",
    "status": "active",
    "reported_at": "2025-12-27T10:05:00Z",
    "updated_at": "2025-12-27T10:15:00Z",
    "location": {
      "corridor": "I-80",
      "direction": "eastbound",
      "milepost": 100.5,
      "cross_street": "Exit 125 (Altoona)",
      "county": "Polk",
      "coordinates": {
        "latitude": 41.5868,
        "longitude": -93.6250
      }
    },
    "impact": {
      "lanes_blocked": "right 2 lanes",
      "lanes_open": "left lane, shoulder",
      "delay_minutes": 35,
      "queue_miles": 3.2,
      "expected_clearance": "2025-12-27T11:45:00Z"
    },
    "description": "Multi-vehicle crash involving semi-truck. Emergency crews on
scene.",
    "traveler_advice": "Use alternate route: US-6 or I-35 to I-80 west of Des
Moines.",
    "cameras": [{
      "id": "CAM-I80-MM100-EB",
      "url": "https://lb.511ia.org/ialb/cameras/routeselect.jsf?
view=state&camera=100"
```

```
    }],
    "related_alerts": ["WEATHER-IA-2025-045"]
  }],
  "road_conditions": [{
    "corridor": "I-80",
    "direction": "both",
    "from_milepost": 100,
    "to_milepost": 105,
    "surface_condition": "wet",
    "visibility": "reduced",
    "temperature_f": 34,
    "weather": "light rain, patchy fog",
    "advisory": "Use caution. Reduce speed in fog."
  }],
  "construction": [{
    "id": "WZ-I80-2025-SUMMER",
    "corridor": "I-80",
    "direction": "eastbound",
    "from_milepost": 110,
    "to_milepost": 115,
    "start_date": "2025-06-01",
    "end_date": "2025-09-30",
    "restrictions": "right lane closed, reduced speed 55 mph",
    "work_type": "pavement resurfacing"
  }]
}
```

Method 3: CAP Alert Distribution

Push alerts to 511 for immediate traveler notification

When to Send CAP:

- Major incidents (severity ≥ major)
- Road closures
- Weather emergencies
- Amber alerts related to highways

Endpoint:

```
POST /api/511/alerts/cap
```

Payload:

```
<alert xmlns="urn:oasis:names:tc:emergency:cap:1.2">
  <identifier>IA-DOT-I80-ALERT-2025-001</identifier>
  <sender>iowa.corridor@iowa.gov</sender>
  <sent>2025-12-27T10:15:00-06:00</sent>
  <status>Actual</status>
  <msgType>Alert</msgType>
  <scope>Public</scope>
```

```
  <info>
    <category>Transport</category>
    <event>Major Traffic Incident</event>
    <urgency>Expected</urgency>
    <severity>Severe</severity>
    <certainty>Observed</certainty>
    <headline>Major Crash I-80 EB at MM 100 - Expect Delays</headline>
    <description>
      Multi-vehicle crash blocking right 2 lanes on I-80 eastbound at
      milepost 100.5 near Altoona. Emergency crews on scene. Expect
      delays of 30-40 minutes. Queue extends back 3+ miles.
    </description>
    <instruction>
      Use alternate routes. Consider US-6 or I-35 to bypass area.
      Reduce speed and watch for emergency vehicles.
    </instruction>
    <contact>Iowa DOT TMC: 515-239-1101</contact>

    <area>
      <areaDesc>I-80 Eastbound MM 98-105, Polk County, Iowa</areaDesc>
      <polygon>
        41.59,-93.65 41.59,-93.55 41.57,-93.55 41.57,-93.65 41.59,-93.65
      </polygon>
    </area>

    <parameter>
      <valueName>EventType</valueName>
      <value>TrafficIncident</value>
    </parameter>
    <parameter>
      <valueName>Corridor</valueName>
      <value>I-80</value>
    </parameter>
    <parameter>
      <valueName>Milepost</valueName>
      <value>100.5</value>
    </parameter>
  </info>
</alert>
```

511 System Actions:

- Post to website homepage
- Send to mobile app subscribers
- Update IVR (phone) system
- Tweet from @511Iowa account
- Email/SMS alert subscribers

Method 4: Subscription-Based Webhooks

511 subscribes to specific event types

Setup:

```
POST /api/511/subscriptions

{
  "subscriber_id": "511-iowa",
  "webhook_url": "https://api.511ia.org/webhooks/corridor-events",
  "webhook_secret": "shared-secret-key",
  "filters": {
    "corridors": ["I-80", "I-35", "I-29"],
    "event_types": ["crash", "road-closed", "weather"],
    "min_severity": "major",
    "counties": ["Polk", "Dallas", "Warren"]
  },
  "delivery_method": "webhook",
  "retry_policy": {
    "max_attempts": 3,
    "backoff": "exponential"
  }
}
```

When event matches, Corridor Communicator POSTs:

```
POST https://api.511ia.org/webhooks/corridor-events

Headers:
  X-Corridor-Signature: sha256=hash-of-body-with-secret
  Content-Type: application/json

Body:
{
  "subscription_id": "sub-511-iowa-001",
  "event_id": "IA-I80-2025-001",
  "event_type": "crash",
  "timestamp": "2025-12-27T10:15:00Z",
  "data": {
    /* Full event details */
  }
}
```

511 System Acknowledges:

```
HTTP/1.1 200 OK
{
  "received": true,
  "event_id": "IA-I80-2025-001",
  "actions_taken": [
    "posted_to_website",
    "sent_to_mobile_apps",
    "updated_ivr"
  ]
}
```

National Event Feed Aggregation

Multi-Source Integration

```
National Event Sources:
├── State DOT TMCs (50 states)
│   ├── TMDD XML Feeds
│   ├── REST APIs
│   └── WebSocket Streams
├── 511 Systems (40+ systems)
│   ├── XML Feeds
│   └── JSON APIs
├── Federal Systems
│   ├── FHWA Real-Time Data
│   ├── NWS Weather Alerts (CAP)
│   └── USGS Earthquake Alerts
├── Commercial Providers
│   ├── INRIX (incidents, speeds)
│   ├── HERE (traffic flow)
│   └── Waze (crowdsourced)
└── Social Media
    ├── Twitter/X (#traffic)
    └── DOT Social Feeds
```

Data Ingestion Pipeline

```
// Event ingestion from multiple sources
class NationalEventAggregator {
  constructor() {
    this.sources = [];
    this.eventQueue = new Queue('national-events');
    this.deduplicator = new EventDeduplicator();
  }

  // Add data source
  addSource(source) {
    const ingestor = {
      id: source.id,
      type: source.type, // 'tmdd', 'cap', 'wzdx', 'json'
      url: source.url,
      poll_interval: source.poll_interval || 60000,
      parser: this.getParser(source.type)
    };

    // Start polling
    setInterval(async () => {
      try {
        const data = await fetch(source.url);
        const events = ingestor.parser(data);
```

```
      for (const event of events) {
        await this.ingestEvent(event, source.id);
      }
    } catch (error) {
      console.error(`Failed to fetch from ${source.id}:`, error);
    }
  }, ingestor.poll_interval);

  this.sources.push(ingestor);
}

// Ingest and process event
async ingestEvent(rawEvent, sourceId) {
  // Normalize to common format
  const normalized = this.normalizeEvent(rawEvent, sourceId);

  // Deduplicate (same event from multiple sources)
  const isDuplicate = await this.deduplicator.check(normalized);
  if (isDuplicate) {
    console.log(`Duplicate event filtered: ${normalized.id}`);
    return;
  }

  // Enrich with additional data
  const enriched = await this.enrichEvent(normalized);

  // Store in database
  await db.run(`
    INSERT INTO national_events
    (event_id, source_id, event_type, severity, corridor,
     latitude, longitude, data, ingested_at)
    VALUES (?, ?, ?, ?, ?, ?, ?, ?, ?)
  `, [
    enriched.id,
    sourceId,
    enriched.type,
    enriched.severity,
    enriched.corridor,
    enriched.location.latitude,
    enriched.location.longitude,
    JSON.stringify(enriched),
    new Date()
  ]);

  // Analyze corridor impact
  await this.analyzeCorridorImpact(enriched);
}

// Normalize different formats to common schema
normalizeEvent(rawEvent, sourceId) {
  const parsers = {
    'tmdd': this.parseTMDD,
```

```
    'cap': this.parseCAP,
    'wzdx': this.parseWZDx,
    'inrix': this.parseINRIX
  };

  const sourceType = this.sources.find(s => s.id === sourceId).type;
  return parsers[sourceType](rawEvent);
}

// Enrich event with corridor context
async enrichEvent(event) {
  // Add corridor information
  if (!event.corridor) {
    event.corridor = await this.geocodeToCorridor(
      event.location.latitude,
      event.location.longitude
    );
  }

  // Add milepost if missing
  if (!event.milepost && event.corridor) {
    event.milepost = await this.calculateMilepost(
      event.corridor,
      event.location.latitude,
      event.location.longitude
    );
  }

  // Add nearby cameras
  event.cameras = await this.findNearbyCameras(
    event.location.latitude,
    event.location.longitude,
    5 // 5 mile radius
  );

  // Add weather data
  event.weather = await this.getWeatherAtLocation(
    event.location.latitude,
    event.location.longitude
  );

  return event;
}

// Analyze impact on corridors
async analyzeCorridorImpact(event) {
  // Get all corridors within impact radius
  const impactedCorridors = await this.getCorridorsInRadius(
    event.location.latitude,
    event.location.longitude,
    event.impact.radius || 10 // miles
  );
```

```
    for (const corridor of impactedCorridors) {
      // Calculate impact score
      const impactScore = this.calculateImpactScore(event, corridor);

      if (impactScore >= NOTIFICATION_THRESHOLD) {
        // Trigger auto-notifications
        await this.triggerNotifications(event, corridor, impactScore);
      }
    }
  }
}
```

Event Deduplication

Challenge: Same incident reported by multiple sources

Solution: Spatial-temporal correlation

```
class EventDeduplicator {
  async check(event) {
    // Find events in same area within time window
    const similar = await db.all(`
      SELECT * FROM national_events
      WHERE event_type = ?
        AND corridor = ?
        AND ABS(latitude - ?) < 0.01  -- ~0.6 miles
        AND ABS(longitude - ?) < 0.01
        AND ingested_at > datetime('now', '-30 minutes')
    `, [
      event.type,
      event.corridor,
      event.location.latitude,
      event.location.longitude
    ]);

    if (similar.length === 0) {
      return false; // Not a duplicate
    }

    // Merge with existing event (take best data from each source)
    const existing = similar[0];
    const merged = this.mergeEvents(existing, event);

    await db.run(`
      UPDATE national_events
      SET data = ?, sources = ?
      WHERE event_id = ?
    `, [
      JSON.stringify(merged),
      JSON.stringify([...existing.sources, event.source]),
      existing.event_id
```

```
    ]);

    return true; // Is a duplicate
  }

  mergeEvents(existing, incoming) {
    return {
      ...existing,
      // Take most recent timestamp
      timestamp: Math.max(existing.timestamp, incoming.timestamp),
      // Take highest severity
      severity: this.maxSeverity(existing.severity, incoming.severity),
      // Combine descriptions
      description: `${existing.description}\n\nUpdate: ${incoming.description}`,
      // Merge impact data (take maximum values)
      impact: {
        lanes_blocked: Math.max(
          existing.impact.lanes_blocked,
          incoming.impact.lanes_blocked
        ),
        delay_minutes: Math.max(
          existing.impact.delay_minutes,
          incoming.impact.delay_minutes
        )
      },
      // Track all sources
      sources: [...new Set([...existing.sources, incoming.source])]
    };
  }
}
```

Auto-Notification Triggers

Trigger Rules Engine

```
class NotificationTriggerEngine {
  constructor() {
    this.rules = [];
    this.loadRules();
  }

  loadRules() {
    // Rule 1: Major incidents on priority corridors
    this.rules.push({
      name: 'Major Incident - Priority Corridors',
      condition: (event, corridor) => {
        return event.severity === 'major' &&
               corridor.priority === 'high' &&
               event.type === 'crash';
      },
```

```
      actions: [
        { type: 'email', recipients: 'tmc-operators' },
        { type: 'sms', recipients: 'on-call-supervisors' },
        { type: 'push', app: 'tmc-mobile' },
        { type: 'vms', message: 'CRASH AHEAD - USE CAUTION' }
      ],
      priority: 'high'
    });

    // Rule 2: Road closures
    this.rules.push({
      name: 'Road Closure Alert',
      condition: (event, corridor) => {
        return event.type === 'road-closed' ||
               (event.impact && event.impact.lanes_blocked >=
event.impact.total_lanes);
      },
      actions: [
        { type: 'email', recipients: 'all-staff' },
        { type: 'sms', recipients: 'supervisors' },
        { type: '511-alert', priority: 'immediate' },
        { type: 'social-media', platforms: ['twitter', 'facebook'] },
        { type: 'vms', message: 'ROAD CLOSED - USE ALT ROUTE' }
      ],
      priority: 'critical'
    });

    // Rule 3: Multi-state corridor events
    this.rules.push({
      name: 'Multi-State Coordination',
      condition: (event, corridor) => {
        return corridor.multi_state === true &&
               event.severity >= 'moderate';
      },
      actions: [
        { type: 'webhook', url: 'https://api.neighboring-state.gov/events' },
        { type: 'email', recipients: 'regional-coordinators' }
      ],
      priority: 'medium'
    });

    // Rule 4: Weather emergencies
    this.rules.push({
      name: 'Severe Weather Alert',
      condition: (event, corridor) => {
        return event.type === 'weather' &&
               (event.subtype === 'blizzard' ||
                event.subtype === 'ice-storm' ||
                event.visibility < 0.25);
      },
      actions: [
        { type: 'email', recipients: 'all-staff' },
```

```
        { type: 'sms', recipients: 'snow-plow-operators' },
        { type: '511-alert', priority: 'immediate' },
        { type: 'cap-alert', scope: 'public' },
        { type: 'vms', message: 'SEVERE WEATHER - REDUCE SPEED' }
      ],
      priority: 'critical'
    });

    // Rule 5: Construction zone incidents
    this.rules.push({
      name: 'Work Zone Incident',
      condition: (event, corridor) => {
        return this.isInWorkZone(event, corridor) &&
               event.type === 'crash';
      },
      actions: [
        { type: 'email', recipients: 'work-zone-supervisors' },
        { type: 'sms', recipients: 'contractor-on-site' },
        { type: 'vms', message: 'INCIDENT IN WORK ZONE' }
      ],
      priority: 'high'
    });

    // Rule 6: Hazmat spills
    this.rules.push({
      name: 'Hazardous Materials Alert',
      condition: (event, corridor) => {
        return event.type === 'hazmat' ||
               (event.description &&
event.description.toLowerCase().includes('spill'));
      },
      actions: [
        { type: 'email', recipients: 'emergency-coordinators' },
        { type: 'sms', recipients: 'hazmat-team' },
        { type: 'phone-call', numbers: 'emergency-contacts' },
        { type: 'cap-alert', scope: 'public', urgency: 'immediate' },
        { type: '511-alert', priority: 'critical' },
        { type: 'vms', message: 'HAZMAT INCIDENT - AREA CLOSED' }
      ],
      priority: 'critical'
    });

    // Rule 7: Recurring congestion (analytics-based)
    this.rules.push({
      name: 'Unexpected Congestion',
      condition: (event, corridor) => {
        return event.type === 'congestion' &&
               event.speed < corridor.typical_speed * 0.5 &&
               !this.isRushHour();
      },
      actions: [
        { type: 'email', recipients: 'traffic-analysts' },
```

```
      { type: 'dashboard-alert', display: 'anomaly-detection' }
    ],
    priority: 'low'
  });
}

async evaluateEvent(event, corridor) {
  const triggeredRules = [];

  for (const rule of this.rules) {
    if (rule.condition(event, corridor)) {
      triggeredRules.push(rule);
      await this.executeActions(rule.actions, event, corridor);

      // Log trigger
      await db.run(`
        INSERT INTO notification_triggers
        (event_id, rule_name, priority, triggered_at)
        VALUES (?, ?, ?, ?)
      `, [event.id, rule.name, rule.priority, new Date()]);
    }
  }

  return triggeredRules;
}

async executeActions(actions, event, corridor) {
  for (const action of actions) {
    try {
      await this.performAction(action, event, corridor);
    } catch (error) {
      console.error(`Failed to execute action ${action.type}:`, error);
      // Continue with other actions
    }
  }
}

async performAction(action, event, corridor) {
  const handlers = {
    'email': this.sendEmail,
    'sms': this.sendSMS,
    'push': this.sendPushNotification,
    'vms': this.postVMSMessage,
    '511-alert': this.send511Alert,
    'cap-alert': this.sendCAPAlert,
    'webhook': this.callWebhook,
    'social-media': this.postSocialMedia,
    'phone-call': this.initiatePhoneCall,
    'dashboard-alert': this.showDashboardAlert
  };

  const handler = handlers[action.type];
```

```
    if (handler) {
      await handler.call(this, action, event, corridor);
    }
  }
}
```

Notification Templates

Email Template:

```
<!DOCTYPE html>
<html>
<head>
  <style>
    .alert-critical { background: #fee; border-left: 4px solid #c00; }
    .alert-high { background: #ffeaa7; border-left: 4px solid #f90; }
    .alert-medium { background: #dfe6e9; border-left: 4px solid #2980b9; }
  </style>
</head>
<body>
  <div class="alert-{{severity}}">
    <h2>{{event_type}} Alert - {{corridor}}</h2>
    <p><strong>Location:</strong> {{corridor}} {{direction}} @ MM {{milepost}}</p>
    <p><strong>Time:</strong> {{timestamp}}</p>
    <p><strong>Severity:</strong> {{severity}}</p>
    <p><strong>Impact:</strong> {{lanes_blocked}} lanes blocked, {{delay}} minute
delay</p>
    <p><strong>Description:</strong> {{description}}</p>

    {{#if cameras}}
    <p><strong>Cameras:</strong></p>
    <ul>
      {{#each cameras}}
      <li><a href="{{url}}">{{name}}</a></li>
      {{/each}}
    </ul>
    {{/if}}

    <p><a href="{{dashboard_url}}">View on TMC Dashboard</a></p>
  </div>
</body>
</html>
```

SMS Template:

```
🚨 {{severity|upper}} ALERT
{{corridor}} {{direction}} MM {{milepost}}
{{event_type}}: {{description|truncate:100}}
{{lanes_blocked}} lanes blocked
Delay: {{delay}} min
More: {{short_url}}
```

Push Notification:

```
{
  "title": "🚨 {{severity}} Alert: {{corridor}}",
  "body": "{{event_type}} @ MM {{milepost}}. {{lanes_blocked}} lanes blocked.
{{delay}} min delay.",
  "data": {
    "event_id": "{{event_id}}",
    "corridor": "{{corridor}}",
    "latitude": {{latitude}},
    "longitude": {{longitude}}
  },
  "priority": "high",
  "sound": "alert_critical.mp3",
  "action": {
    "type": "open_map",
    "coordinates": [{{latitude}}, {{longitude}}]
  }
}
```

Implementation Guide

Backend Setup

Step 1: Install Dependencies

```
npm install --save \
  xml2js \          # Parse TMDD/CAP/EDXL XML
  fast-xml-parser \ # Faster XML parsing
  ws \              # WebSocket server
  axios \           # HTTP requests
  nodemailer \      # Email notifications
  twilio \          # SMS notifications
  node-pushnotifications \ # Mobile push
  turf \            # Geospatial calculations
  cron \            # Scheduled tasks
  bull \            # Job queue for processing
  ioredis           # Redis for caching/queues
```

Step 2: Database Schema

```
-- National events table
CREATE TABLE national_events (
  id SERIAL PRIMARY KEY,
  event_id VARCHAR(100) UNIQUE NOT NULL,
  source_id VARCHAR(100) NOT NULL,
  event_type VARCHAR(50) NOT NULL,
  severity VARCHAR(20),
  status VARCHAR(20) DEFAULT 'active',
```

```
  corridor VARCHAR(50),
  direction VARCHAR(20),
  milepost DECIMAL(6,2),
  latitude DECIMAL(10,6),
  longitude DECIMAL(10,6),
  description TEXT,
  impact_data JSONB,
  raw_data JSONB,
  sources TEXT[], -- Array of source IDs
  ingested_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
  updated_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
  cleared_at TIMESTAMP,
  INDEX idx_corridor (corridor),
  INDEX idx_location (latitude, longitude),
  INDEX idx_timestamp (ingested_at),
  INDEX idx_status (status)
);

-- Data sources configuration
CREATE TABLE event_sources (
  id SERIAL PRIMARY KEY,
  source_id VARCHAR(100) UNIQUE NOT NULL,
  name VARCHAR(200),
  type VARCHAR(50), -- 'tmdd', 'cap', 'wzdx', 'json'
  url TEXT,
  poll_interval INTEGER DEFAULT 60, -- seconds
  authentication JSONB,
  filters JSONB,
  enabled BOOLEAN DEFAULT true,
  last_poll TIMESTAMP,
  last_success TIMESTAMP,
  error_count INTEGER DEFAULT 0,
  created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);

-- Notification subscriptions
CREATE TABLE notification_subscriptions (
  id SERIAL PRIMARY KEY,
  subscriber_id VARCHAR(100) NOT NULL,
  subscription_type VARCHAR(50), -- 'email', 'sms', 'webhook', 'push'
  delivery_target TEXT NOT NULL, -- email, phone, URL, device token
  filters JSONB, -- Corridors, event types, severity
  enabled BOOLEAN DEFAULT true,
  created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
  INDEX idx_subscriber (subscriber_id),
  INDEX idx_type (subscription_type)
);

-- Notification log
CREATE TABLE notification_log (
  id SERIAL PRIMARY KEY,
  event_id VARCHAR(100) NOT NULL,
```

```
  subscription_id INTEGER REFERENCES notification_subscriptions(id),
  notification_type VARCHAR(50),
  delivery_status VARCHAR(20), -- 'sent', 'failed', 'pending'
  delivery_attempts INTEGER DEFAULT 0,
  error_message TEXT,
  sent_at TIMESTAMP,
  delivered_at TIMESTAMP,
  INDEX idx_event (event_id),
  INDEX idx_status (delivery_status)
);

-- TMC/511 integration endpoints
CREATE TABLE integration_endpoints (
  id SERIAL PRIMARY KEY,
  endpoint_id VARCHAR(100) UNIQUE NOT NULL,
  system_type VARCHAR(50), -- 'tmc', '511', 'vms', 'fleet'
  name VARCHAR(200),
  organization VARCHAR(200),
  endpoint_url TEXT,
  webhook_url TEXT,
  api_key_hash TEXT,
  subscription_filters JSONB,
  enabled BOOLEAN DEFAULT true,
  last_delivery TIMESTAMP,
  delivery_success_count INTEGER DEFAULT 0,
  delivery_failure_count INTEGER DEFAULT 0,
  created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);
```

Step 3: Create API Endpoints

Add to `backend_proxy_server.js`:

```
const xml2js = require('xml2js');
const WebSocket = require('ws');
const turf = require('@turf/turf');

// ============================================================================
// TMC INTEGRATION ENDPOINTS
// ============================================================================

// TMDD XML Feed - TMC pulls events
app.get('/api/tmc/events/tmdd', async (req, res) => {
  try {
    const { corridor, severity, event_type, since, bbox } = req.query;

    let query = `
      SELECT * FROM national_events
      WHERE status = 'active'
    `;
    const params = [];
```

```
    if (corridor) {
      query += ` AND corridor = ?`;
      params.push(corridor);
    }

    if (severity) {
      query += ` AND severity = ?`;
      params.push(severity);
    }

    if (event_type) {
      query += ` AND event_type = ?`;
      params.push(event_type);
    }

    if (since) {
      query += ` AND updated_at >= ?`;
      params.push(since);
    }

    if (bbox) {
      const [minLon, minLat, maxLon, maxLat] = bbox.split(',').map(parseFloat);
      query += ` AND latitude BETWEEN ? AND ? AND longitude BETWEEN ? AND ?`;
      params.push(minLat, maxLat, minLon, maxLon);
    }

    query += ` ORDER BY ingested_at DESC LIMIT 500`;

    const events = db.db.prepare(query).all(...params);

    // Build TMDD XML
    const tmddXml = buildTMDDFeed(events);

    res.set('Content-Type', 'application/xml');
    res.send(tmddXml);

  } catch (error) {
    console.error('Error generating TMDD feed:', error);
    res.status(500).json({ error: 'Failed to generate TMDD feed' });
  }
});

// WebSocket stream for real-time events
const wss = new WebSocket.Server({ noServer: true });

wss.on('connection', (ws) => {
  console.log('TMC WebSocket client connected');

  ws.subscriptions = {
    corridors: [],
    event_types: [],
    min_severity: null
```

```
  };

  ws.on('message', (message) => {
    try {
      const data = JSON.parse(message);

      if (data.action === 'subscribe') {
        ws.subscriptions = {
          corridors: data.corridors || [],
          event_types: data.event_types || [],
          min_severity: data.min_severity || null
        };

        ws.send(JSON.stringify({
          status: 'subscribed',
          subscriptions: ws.subscriptions
        }));
      }
    } catch (error) {
      console.error('WebSocket message error:', error);
    }
  });

  ws.on('close', () => {
    console.log('TMC WebSocket client disconnected');
  });
});

// Upgrade HTTP to WebSocket
app.server.on('upgrade', (request, socket, head) => {
  if (request.url === '/api/tmc/stream') {
    wss.handleUpgrade(request, socket, head, (ws) => {
      wss.emit('connection', ws, request);
    });
  }
});

// Broadcast event to WebSocket clients
function broadcastEventToTMC(event) {
  wss.clients.forEach((client) => {
    if (client.readyState === WebSocket.OPEN) {
      const subs = client.subscriptions;

      // Check if event matches client subscriptions
      const corridorMatch = subs.corridors.length === 0 ||
                            subs.corridors.includes(event.corridor);
      const typeMatch = subs.event_types.length === 0 ||
                        subs.event_types.includes(event.event_type);
      const severityMatch = !subs.min_severity ||
                            compareSeverity(event.severity, subs.min_severity) >= 0;

      if (corridorMatch && typeMatch && severityMatch) {
```

```
        client.send(JSON.stringify({
          message_type: 'event_update',
          ...event
        }));
      }
    }
  });
}

// Post message to VMS board (NTCIP 1218)
app.post('/api/tmc/vms/post-message', async (req, res) => {
  try {
    const { equipment_id, message } = req.body;

    // Get VMS equipment details
    const vms = db.db.prepare(`
      SELECT * FROM its_equipment
      WHERE equipment_id = ? AND equipment_type = 'vms'
    `).get(equipment_id);

    if (!vms) {
      return res.status(404).json({ error: 'VMS not found' });
    }

    // Convert message to NTCIP 1218 MULTI string
    const multiString = buildNTCIPMessage(message.pages);

    // Send to VMS controller (requires NTCIP library)
    // This is a placeholder - actual implementation depends on VMS vendor
    const result = await sendNTCIPCommand(vms, {
      oid: 'dmsMessageMultiString.1.1.0',
      value: multiString,
      priority: message.priority === 'high' ? 255 : 128,
      duration: message.duration || 3600
    });

    // Log the message posting
    await db.run(`
      INSERT INTO vms_message_log
      (equipment_id, message_text, priority, duration, triggered_by, posted_at)
      VALUES (?, ?, ?, ?, ?, ?)
    `, [
      equipment_id,
      JSON.stringify(message),
      message.priority,
      message.duration,
      message.triggered_by,
      new Date()
    ]);

    res.json({ success: true, result });
```

```
  } catch (error) {
    console.error('Error posting VMS message:', error);
    res.status(500).json({ error: 'Failed to post VMS message' });
  }
});

// Get events as GeoJSON map layer
app.get('/api/tmc/events/map-layer', async (req, res) => {
  try {
    const { corridor, severity } = req.query;

    let query = `SELECT * FROM national_events WHERE status = 'active'`;
    const params = [];

    if (corridor) {
      query += ` AND corridor = ?`;
      params.push(corridor);
    }

    if (severity) {
      query += ` AND severity = ?`;
      params.push(severity);
    }

    const events = db.db.prepare(query).all(...params);

    // Convert to GeoJSON
    const geojson = {
      type: 'FeatureCollection',
      features: events.map(event => ({
        type: 'Feature',
        geometry: {
          type: 'Point',
          coordinates: [event.longitude, event.latitude]
        },
        properties: {
          event_id: event.event_id,
          event_type: event.event_type,
          severity: event.severity,
          corridor: event.corridor,
          milepost: event.milepost,
          description: event.description,
          icon: getEventIcon(event.event_type, event.severity),
          popup_html: buildPopupHTML(event),
          style: getEventStyle(event.severity)
        }
      }))
    };

    res.json(geojson);

  } catch (error) {
```

```
    console.error('Error generating map layer:', error);
    res.status(500).json({ error: 'Failed to generate map layer' });
  }
});

// ============================================================================
// 511 SYSTEM INTEGRATION ENDPOINTS
// ============================================================================

// XML Event Feed (SAE J2354)
app.get('/api/511/events/xml', async (req, res) => {
  try {
    const { state, corridor } = req.query;

    let query = `SELECT * FROM national_events WHERE status = 'active'`;
    const params = [];

    if (corridor) {
      query += ` AND corridor = ?`;
      params.push(corridor);
    }

    const events = db.db.prepare(query).all(...params);

    // Build SAE J2354 XML
    const xmlFeed = build511XMLFeed(events, state);

    res.set('Content-Type', 'application/xml');
    res.send(xmlFeed);

  } catch (error) {
    console.error('Error generating 511 XML feed:', error);
    res.status(500).json({ error: 'Failed to generate 511 feed' });
  }
});

// JSON Event Feed
app.get('/api/511/events/json', async (req, res) => {
  try {
    const { corridor, format = 'v2' } = req.query;

    let query = `SELECT * FROM national_events WHERE status = 'active'`;
    const params = [];

    if (corridor) {
      query += ` AND corridor = ?`;
      params.push(corridor);
    }

    const events = db.db.prepare(query).all(...params);

    // Get related data
```

```
    const incidents = events.filter(e => e.event_type === 'crash' || e.event_type
=== 'hazard');
    const roadConditions = events.filter(e => e.event_type === 'weather');
    const construction = db.db.prepare(`
      SELECT * FROM work_zones WHERE status = 'active'
    `).all();

    const feed = {
      metadata: {
        version: format,
        provider: 'DOT Corridor Communicator',
        timestamp: new Date().toISOString(),
        update_frequency: 60,
        coverage: {
          corridors: [...new Set(events.map(e => e.corridor))]
        }
      },
      incidents: incidents.map(e => format511Incident(e)),
      road_conditions: roadConditions.map(e => format511RoadCondition(e)),
      construction: construction.map(wz => format511Construction(wz))
    };

    res.json(feed);

  } catch (error) {
    console.error('Error generating 511 JSON feed:', error);
    res.status(500).json({ error: 'Failed to generate 511 feed' });
  }
});

// CAP Alert Distribution
app.post('/api/511/alerts/cap', async (req, res) => {
  try {
    const { event_id } = req.body;

    const event = db.db.prepare(`
      SELECT * FROM national_events WHERE event_id = ?
    `).get(event_id);

    if (!event) {
      return res.status(404).json({ error: 'Event not found' });
    }

    // Only send CAP for major/critical events
    if (event.severity !== 'major' && event.severity !== 'critical') {
      return res.status(400).json({ error: 'Event severity too low for CAP alert'
});
    }

    // Build CAP alert
    const capAlert = buildCAPAlert(event);
```

```
    // Distribute to subscribed 511 systems
    const endpoints = db.db.prepare(`
      SELECT * FROM integration_endpoints
      WHERE system_type = '511'
        AND enabled = true
        AND subscription_filters->>'cap_alerts' = 'true'
    `).all();

    const results = [];
    for (const endpoint of endpoints) {
      try {
        const response = await axios.post(endpoint.webhook_url, capAlert, {
          headers: {
            'Content-Type': 'application/xml',
            'X-Signature': generateSignature(capAlert, endpoint.api_key_hash)
          }
        });

        results.push({
          endpoint: endpoint.name,
          status: 'delivered',
          response: response.data
        });

      } catch (error) {
        results.push({
          endpoint: endpoint.name,
          status: 'failed',
          error: error.message
        });
      }
    }

    res.json({ success: true, alert_id: event.event_id, deliveries: results });

  } catch (error) {
    console.error('Error distributing CAP alert:', error);
    res.status(500).json({ error: 'Failed to distribute CAP alert' });
  }
});

// Webhook Subscriptions
app.post('/api/511/subscriptions', async (req, res) => {
  try {
    const { subscriber_id, webhook_url, webhook_secret, filters, delivery_method,
retry_policy } = req.body;

    // Create endpoint subscription
    const result = await db.run(`
      INSERT INTO integration_endpoints
      (endpoint_id, system_type, webhook_url, api_key_hash, subscription_filters)
      VALUES (?, ?, ?, ?, ?)
```

```
    `, [
      `sub-511-${subscriber_id}-${Date.now()}`,
      '511',
      webhook_url,
      hashSecret(webhook_secret),
      JSON.stringify(filters)
    ]);

    res.json({
      success: true,
      subscription_id: result.lastID,
      message: 'Subscription created successfully'
    });

  } catch (error) {
    console.error('Error creating subscription:', error);
    res.status(500).json({ error: 'Failed to create subscription' });
  }
});

// ============================================================================
// NATIONAL EVENT AGGREGATION ENDPOINTS
// ============================================================================

// Ingest event from external source
app.post('/api/events/ingest', async (req, res) => {
  try {
    const { source_id, format, data } = req.body;

    // Verify source is authorized
    const source = db.db.prepare(`
      SELECT * FROM event_sources WHERE source_id = ? AND enabled = true
    `).get(source_id);

    if (!source) {
      return res.status(401).json({ error: 'Unauthorized source' });
    }

    // Parse event based on format
    let events = [];
    switch (format) {
      case 'tmdd':
        events = parseTMDD(data);
        break;
      case 'cap':
        events = parseCAP(data);
        break;
      case 'wzdx':
        events = parseWZDx(data);
        break;
      case 'json':
        events = [data];
```

```
        break;
      default:
        return res.status(400).json({ error: 'Unsupported format' });
    }

    // Process each event
    const ingested = [];
    for (const rawEvent of events) {
      const normalized = normalizeEvent(rawEvent, source_id);
      const enriched = await enrichEvent(normalized);

      // Check for duplicates
      const isDuplicate = await checkDuplicate(enriched);
      if (!isDuplicate) {
        await storeEvent(enriched);
        await analyzeCorridorImpact(enriched);
        broadcastEventToTMC(enriched);
        ingested.push(enriched.event_id);
      }
    }

    res.json({
      success: true,
      ingested_count: ingested.length,
      event_ids: ingested
    });

  } catch (error) {
    console.error('Error ingesting events:', error);
    res.status(500).json({ error: 'Failed to ingest events' });
  }
});

// Get national event feed
app.get('/api/events/national', async (req, res) => {
  try {
    const { corridor, state, event_type, severity, limit = 100 } = req.query;

    let query = `SELECT * FROM national_events WHERE status = 'active'`;
    const params = [];

    if (corridor) {
      query += ` AND corridor = ?`;
      params.push(corridor);
    }

    if (event_type) {
      query += ` AND event_type = ?`;
      params.push(event_type);
    }

    if (severity) {
```

```
      query += ` AND severity = ?`;
      params.push(severity);
    }

    query += ` ORDER BY ingested_at DESC LIMIT ?`;
    params.push(parseInt(limit));

    const events = db.db.prepare(query).all(...params);

    res.json({
      success: true,
      count: events.length,
      events: events.map(e => ({
        ...e,
        impact_data: JSON.parse(e.impact_data || '{}'),
        sources: JSON.parse(e.sources || '[]')
      }))
    });

  } catch (error) {
    console.error('Error fetching national events:', error);
    res.status(500).json({ error: 'Failed to fetch events' });
  }
});

// ============================================================================
// AUTO-NOTIFICATION ENDPOINTS
// ============================================================================

// Create notification subscription
app.post('/api/notifications/subscribe', async (req, res) => {
  try {
    const { subscriber_id, subscription_type, delivery_target, filters } = req.body;

    const result = await db.run(`
      INSERT INTO notification_subscriptions
      (subscriber_id, subscription_type, delivery_target, filters)
      VALUES (?, ?, ?, ?)
    `, [
      subscriber_id,
      subscription_type,
      delivery_target,
      JSON.stringify(filters)
    ]);

    res.json({
      success: true,
      subscription_id: result.lastID,
      message: 'Notification subscription created'
    });

  } catch (error) {
```

```
    console.error('Error creating subscription:', error);
    res.status(500).json({ error: 'Failed to create subscription' });
  }
});

// Trigger notifications for event
async function triggerNotifications(event, corridor, impactScore) {
  try {
    // Get matching subscriptions
    const subscriptions = db.db.prepare(`
      SELECT * FROM notification_subscriptions
      WHERE enabled = true
    `).all();

    for (const sub of subscriptions) {
      const filters = JSON.parse(sub.filters || '{}');

      // Check if event matches subscription filters
      const corridorMatch = !filters.corridors ||
filters.corridors.includes(event.corridor);
      const typeMatch = !filters.event_types ||
filters.event_types.includes(event.event_type);
      const severityMatch = !filters.min_severity ||
                            compareSeverity(event.severity, filters.min_severity) >=
0;

      if (corridorMatch && typeMatch && severityMatch) {
        await sendNotification(sub, event);
      }
    }

  } catch (error) {
    console.error('Error triggering notifications:', error);
  }
}

// Send notification
async function sendNotification(subscription, event) {
  try {
    let result;

    switch (subscription.subscription_type) {
      case 'email':
        result = await sendEmailNotification(subscription.delivery_target, event);
        break;
      case 'sms':
        result = await sendSMSNotification(subscription.delivery_target, event);
        break;
      case 'webhook':
        result = await sendWebhookNotification(subscription.delivery_target, event);
        break;
      case 'push':
```

```
        result = await sendPushNotification(subscription.delivery_target, event);
        break;
    }

    // Log notification
    await db.run(`
      INSERT INTO notification_log
      (event_id, subscription_id, notification_type, delivery_status, sent_at)
      VALUES (?, ?, ?, ?, ?)
    `, [
      event.event_id,
      subscription.id,
      subscription.subscription_type,
      result.success ? 'sent' : 'failed',
      new Date()
    ]);

  } catch (error) {
    console.error(`Error sending ${subscription.subscription_type} notification:`,
error);
  }
}

// ============================================================================
// HELPER FUNCTIONS
// ============================================================================

function buildTMDDFeed(events) {
  const builder = new xml2js.Builder({ rootName: 'tmdd:eventFeed' });

  const tmddData = {
    'feed-metadata': {
      publisher: 'DOT Corridor Communicator',
      'update-frequency': 60,
      'last-updated': new Date().toISOString()
    },
    events: {
      event: events.map(e => ({
        'event-id': e.event_id,
        'event-type': e.event_type,
        severity: e.severity,
        location: {
          corridor: e.corridor,
          direction: e.direction,
          milepost: e.milepost,
          latitude: e.latitude,
          longitude: e.longitude
        },
        description: e.description,
        timestamp: e.ingested_at
      }))
    }
```

```
  };

  return builder.buildObject(tmddData);
}

function build511XMLFeed(events, state) {
  const builder = new xml2js.Builder({ rootName: 'traffic-data' });

  const feedData = {
    metadata: {
      provider: `${state} DOT Corridor Communicator`,
      timestamp: new Date().toISOString(),
      'coverage-area': {
        state: state,
        corridors: [...new Set(events.map(e => e.corridor))].join(', ')
      }
    },
    incidents: {
      incident: events.map(e => ({
        id: e.event_id,
        type: e.event_type,
        severity: e.severity,
        'reported-time': e.ingested_at,
        location: {
          route: e.corridor,
          direction: e.direction,
          latitude: e.latitude,
          longitude: e.longitude
        },
        description: e.description
      }))
    }
  };

  return builder.buildObject(feedData);
}

function buildCAPAlert(event) {
  const builder = new xml2js.Builder({ rootName: 'alert' });

  const capData = {
    $: { xmlns: 'urn:oasis:names:tc:emergency:cap:1.2' },
    identifier: `CAP-${event.event_id}`,
    sender: 'corridor.communicator@dot.gov',
    sent: new Date().toISOString(),
    status: 'Actual',
    msgType: 'Alert',
    scope: 'Public',
    info: {
      category: 'Transport',
      event: `${event.event_type} Alert`,
      urgency: event.severity === 'critical' ? 'Immediate' : 'Expected',
```

```
      severity: event.severity === 'critical' ? 'Severe' : 'Moderate',
      certainty: 'Observed',
      headline: `${event.event_type} on ${event.corridor}`,
      description: event.description,
      area: {
        areaDesc: `${event.corridor} ${event.direction} @ MM ${event.milepost}`,
        circle: `${event.latitude},${event.longitude} 5`
      }
    }
  };

  return builder.buildObject(capData);
}

function format511Incident(event) {
  const impactData = JSON.parse(event.impact_data || '{}');

  return {
    id: event.event_id,
    type: event.event_type,
    severity: event.severity,
    status: event.status,
    reported_at: event.ingested_at,
    updated_at: event.updated_at,
    location: {
      corridor: event.corridor,
      direction: event.direction,
      milepost: event.milepost,
      coordinates: {
        latitude: event.latitude,
        longitude: event.longitude
      }
    },
    impact: impactData,
    description: event.description
  };
}

function normalizeEvent(rawEvent, sourceId) {
  // Normalize different event formats to common schema
  return {
    event_id: rawEvent.id || `EVT-${Date.now()}`,
    source_id: sourceId,
    event_type: rawEvent.type || rawEvent.event_type,
    severity: rawEvent.severity || 'moderate',
    corridor: rawEvent.corridor || rawEvent.route,
    direction: rawEvent.direction,
    milepost: rawEvent.milepost,
    location: {
      latitude: rawEvent.latitude || rawEvent.location?.latitude,
      longitude: rawEvent.longitude || rawEvent.location?.longitude
    },
```

```
    description: rawEvent.description,
    impact: rawEvent.impact || {},
    timestamp: rawEvent.timestamp || new Date()
  };
}

async function enrichEvent(event) {
  // Add corridor information if missing
  if (!event.corridor && event.location) {
    event.corridor = await geocodeToCorridor(
      event.location.latitude,
      event.location.longitude
    );
  }

  // Add nearby cameras
  const cameras = db.db.prepare(`
    SELECT * FROM its_equipment
    WHERE equipment_type = 'camera'
      AND ABS(latitude - ?) < 0.05
      AND ABS(longitude - ?) < 0.05
    LIMIT 5
  `).all(event.location.latitude, event.location.longitude);

  event.cameras = cameras.map(c => ({
    id: c.equipment_id,
    name: c.name,
    url: c.stream_url
  }));

  return event;
}

async function checkDuplicate(event) {
  const similar = db.db.prepare(`
    SELECT * FROM national_events
    WHERE event_type = ?
      AND corridor = ?
      AND ABS(latitude - ?) < 0.01
      AND ABS(longitude - ?) < 0.01
      AND ingested_at > datetime('now', '-30 minutes')
  `).get(
    event.event_type,
    event.corridor,
    event.location.latitude,
    event.location.longitude
  );

  return similar !== undefined;
}

async function storeEvent(event) {
```

```
  await db.run(`
    INSERT INTO national_events
    (event_id, source_id, event_type, severity, corridor, direction, milepost,
     latitude, longitude, description, impact_data, sources)
    VALUES (?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?)
  `, [
    event.event_id,
    event.source_id,
    event.event_type,
    event.severity,
    event.corridor,
    event.direction,
    event.milepost,
    event.location.latitude,
    event.location.longitude,
    event.description,
    JSON.stringify(event.impact),
    JSON.stringify([event.source_id])
  ]);
}

async function analyzeCorridorImpact(event) {
  // Get corridors within impact radius
  const corridors = db.db.prepare(`
    SELECT DISTINCT corridor FROM its_equipment
    WHERE ABS(latitude - ?) < 0.1
      AND ABS(longitude - ?) < 0.1
  `).all(event.location.latitude, event.location.longitude);

  for (const corridor of corridors) {
    const impactScore = calculateImpactScore(event, corridor);

    if (impactScore >= 50) {
      await triggerNotifications(event, corridor.corridor, impactScore);
    }
  }
}

function calculateImpactScore(event, corridor) {
  let score = 0;

  // Severity weight
  const severityWeights = { minor: 20, moderate: 40, major: 70, critical: 100 };
  score += severityWeights[event.severity] || 0;

  // Event type weight
  const typeWeights = { crash: 30, 'road-closed': 50, weather: 20, hazmat: 50 };
  score += typeWeights[event.event_type] || 0;

  // Normalize to 0-100
  return Math.min(100, score);
}
```

```
function compareSeverity(severity1, severity2) {
  const levels = { minor: 1, moderate: 2, major: 3, critical: 4 };
  return (levels[severity1] || 0) - (levels[severity2] || 0);
}

function getEventIcon(eventType, severity) {
  const icons = {
    crash: severity === 'critical' ? 'crash-critical' : 'crash-major',
    weather: 'weather-alert',
    'road-closed': 'road-closed',
    hazmat: 'hazmat-warning',
    construction: 'construction-zone'
  };
  return icons[eventType] || 'event-generic';
}

function getEventStyle(severity) {
  const styles = {
    critical: { color: '#FF0000', size: 'large', blink: true },
    major: { color: '#FF6600', size: 'medium', blink: false },
    moderate: { color: '#FFAA00', size: 'small', blink: false },
    minor: { color: '#FFFF00', size: 'small', blink: false }
  };
  return styles[severity] || styles.moderate;
}

function buildPopupHTML(event) {
  return `
    <strong>${event.event_type.toUpperCase()}</strong><br>
    ${event.corridor} ${event.direction} @ MM ${event.milepost}<br>
    <em>${event.description}</em>
  `;
}
```

Step 4: Event Processing Worker

Create `workers/event-processor.js`:

```
const cron = require('node-cron');
const axios = require('axios');

class EventProcessor {
  constructor(db) {
    this.db = db;
    this.sources = [];
    this.init();
  }

  async init() {
    // Load configured event sources
    this.sources = this.db.prepare(`
```

```
    SELECT * FROM event_sources WHERE enabled = true
  `).all();

  // Start polling each source
  for (const source of this.sources) {
    this.startPolling(source);
  }

  // Cleanup old events (every hour)
  cron.schedule('0 * * * *', () => {
    this.cleanupOldEvents();
  });
}

startPolling(source) {
  const interval = source.poll_interval * 1000;

  setInterval(async () => {
    try {
      console.log(`Polling ${source.name}...`);

      const response = await axios.get(source.url, {
        headers: this.buildAuthHeaders(source),
        timeout: 30000
      });

      const events = this.parseEvents(response.data, source.type);
      console.log(`Found ${events.length} events from ${source.name}`);

      for (const event of events) {
        await this.processEvent(event, source.source_id);
      }

      // Update last poll time
      await this.db.run(`
        UPDATE event_sources
        SET last_poll = ?, last_success = ?, error_count = 0
        WHERE source_id = ?
      `, [new Date(), new Date(), source.source_id]);

    } catch (error) {
      console.error(`Error polling ${source.name}:`, error.message);

      // Increment error count
      await this.db.run(`
        UPDATE event_sources
        SET last_poll = ?, error_count = error_count + 1
        WHERE source_id = ?
      `, [new Date(), source.source_id]);
    }
  }, interval);
}
```

```
  parseEvents(data, sourceType) {
    switch (sourceType) {
      case 'tmdd':
        return this.parseTMDD(data);
      case 'cap':
        return this.parseCAP(data);
      case 'wzdx':
        return this.parseWZDx(data);
      case 'json':
        return Array.isArray(data) ? data : [data];
      default:
        return [];
    }
  }

  async processEvent(rawEvent, sourceId) {
    // Normalize, deduplicate, enrich, store
    // (Use functions from backend_proxy_server.js)
  }

  async cleanupOldEvents() {
    // Delete events older than 7 days
    await this.db.run(`
      DELETE FROM national_events
      WHERE status = 'cleared'
        AND cleared_at < datetime('now', '-7 days')
    `);

    console.log('Cleaned up old events');
  }

  buildAuthHeaders(source) {
    const auth = JSON.parse(source.authentication || '{}');
    const headers = {};

    if (auth.type === 'api_key') {
      headers['X-API-Key'] = auth.key;
    } else if (auth.type === 'bearer') {
      headers['Authorization'] = `Bearer ${auth.token}`;
    }

    return headers;
  }
}

module.exports = EventProcessor;
```

API Reference

TMC Integration

GET /api/tmc/events/tmdd

Retrieve events in TMDD XML format.

Query Parameters:

- `corridor` - Filter by corridor (e.g., I-80)
- `severity` - Filter by severity (minor, moderate, major, critical)
- `event_type` - Filter by type (crash, weather, construction, etc.)
- `since` - Events since ISO timestamp
- `bbox` - Bounding box (minLon,minLat,maxLon,maxLat)

Response: TMDD XML feed

WebSocket /api/tmc/stream

Real-time event stream via WebSocket.

Subscribe Message:

```
{
  "action": "subscribe",
  "corridors": ["I-80", "I-35"],
  "event_types": ["crash", "weather"],
  "min_severity": "major"
}
```

Event Message:

```
{
  "message_type": "event_update",
  "event_id": "IA-I80-2025-001",
  "event_type": "crash",
  "severity": "major",
  "location": { ... },
  "impact": { ... }
}
```

POST /api/tmc/vms/post-message

Post message to VMS board.

Request:

```
{
  "equipment_id": "VMS-I80-MM95-EB",
  "message": {
    "pages": [
      { "text": "CRASH AHEAD", "display_time": 3 },
      { "text": "USE CAUTION", "display_time": 2 }
    ],
```

```
    "priority": "high",
    "duration": 3600,
    "triggered_by": "event:IA-I80-2025-001"
  }
}
```

Response:

```
{
  "success": true,
  "result": { ... }
}
```

---

511 System Integration

GET /api/511/events/xml

SAE J2354 compliant XML feed.

Query Parameters:

- `state` - Filter by state
- `corridor` - Filter by corridor

Response: SAE J2354 XML

GET /api/511/events/json

JSON event feed.

Query Parameters:

- `corridor` - Filter by corridor
- `format` - Version (v1, v2)

Response:

```
{
  "metadata": { ... },
  "incidents": [ ... ],
  "road_conditions": [ ... ],
  "construction": [ ... ]
}
```

POST /api/511/alerts/cap

Distribute CAP alert to 511 systems.

Request:

```
{
  "event_id": "IA-I80-2025-001"
```

```
}
```

Response:

```
{
  "success": true,
  "alert_id": "CAP-IA-I80-2025-001",
  "deliveries": [
    { "endpoint": "511 Iowa", "status": "delivered" }
  ]
}
```

POST /api/511/subscriptions

Create webhook subscription.

Request:

```
{
  "subscriber_id": "511-iowa",
  "webhook_url": "https://api.511ia.org/webhooks/events",
  "webhook_secret": "shared-secret",
  "filters": {
    "corridors": ["I-80"],
    "event_types": ["crash", "road-closed"],
    "min_severity": "major"
  }
}
```

Response:

```
{
  "success": true,
  "subscription_id": 123
}
```

National Event Aggregation

POST /api/events/ingest

Ingest events from external sources.

Request:

```
{
  "source_id": "iowa-tmc",
  "format": "tmdd",
  "data": "<tmdd:event>...</tmdd:event>"
}
```

Response:

```
{
  "success": true,
  "ingested_count": 5,
  "event_ids": ["EVT-001", "EVT-002", ...]
}
```

GET /api/events/national

Get aggregated national events.

Query Parameters:

- `corridor` - Filter by corridor
- `state` - Filter by state
- `event_type` - Filter by type
- `severity` - Filter by severity
- `limit` - Max results (default 100)

Response:

```
{
  "success": true,
  "count": 42,
  "events": [ ... ]
}
```

Auto-Notifications

POST /api/notifications/subscribe

Subscribe to event notifications.

Request:

```
{
  "subscriber_id": "user-123",
  "subscription_type": "email",
  "delivery_target": "recipient@yourdot.gov",
  "filters": {
    "corridors": ["I-80"],
    "min_severity": "major"
  }
}
```

Response:

```
{
  "success": true,
```

```
  "subscription_id": 456
}
```

Use Cases & Examples

Use Case 1: Multi-State Corridor Event Sharing

Scenario: Iowa DOT wants to share I-80 incidents with Nebraska DOT in real-time.

Implementation:

```
// Iowa DOT publishes events via TMDD feed
// Nebraska DOT subscribes to Iowa events

// 1. Nebraska configures event source
await axios.post('https://corridor.nebraska.gov/api/admin/sources', {
  source_id: 'iowa-i80',
  name: 'Iowa DOT I-80 Events',
  type: 'tmdd',
  url: 'https://corridor.iowa.gov/api/tmc/events/tmdd?corridor=I-80',
  poll_interval: 60,
  authentication: {
    type: 'api_key',
    key: 'shared-api-key'
  },
  enabled: true
});

// 2. Events automatically flow Iowa → Nebraska
// Iowa incident detected at MM 5 (near IA/NE border)
const iowaEvent = {
  event_id: 'IA-I80-2025-001',
  event_type: 'crash',
  corridor: 'I-80',
  milepost: 5,
  severity: 'major'
};

// 3. Nebraska ingests and enriches with local data
// 4. Nebraska TMC displays on map
// 5. Nebraska posts VMS message: "CRASH IN IOWA - DELAYS EXPECTED"
```

Benefits:

- Travelers get advance warning before crossing state line
- Nebraska can adjust signal timing to accommodate diverted traffic
- Coordinated incident response between states

Use Case 2: 511 System Auto-Update

Scenario: When major incident occurs, automatically push to state 511 system.

Implementation:

```
// 1. Configure 511 webhook subscription
await axios.post('https://corridor.iowa.gov/api/511/subscriptions', {
  subscriber_id: '511-iowa',
  webhook_url: 'https://api.511ia.org/webhooks/corridor-events',
  webhook_secret: 'shared-secret-key',
  filters: {
    corridors: ['I-80', 'I-35', 'I-29'],
    event_types: ['crash', 'road-closed', 'weather'],
    min_severity: 'major'
  }
});

// 2. Major crash occurs on I-80
const event = {
  event_id: 'IA-I80-2025-001',
  event_type: 'crash',
  severity: 'major',
  corridor: 'I-80',
  milepost: 100.5
};

// 3. Corridor Communicator triggers notification
// POST to https://api.511ia.org/webhooks/corridor-events
{
  "subscription_id": "sub-511-iowa-001",
  "event_id": "IA-I80-2025-001",
  "event_type": "crash",
  "data": { /* full event details */ }
}

// 4. 511 System automatically:
// - Updates website homepage
// - Sends mobile app push notifications
// - Updates IVR (phone) system messages
// - Tweets from @511Iowa
// - Emails/SMS subscribers in affected area
```

Result: Travelers receive alerts within seconds of incident detection

---

Use Case 3: National Freight Corridor Monitoring

Scenario: Track incidents across I-80 freight corridor (CA to NJ) and alert fleet operators.

Implementation:

```
// 1. Aggregate events from all I-80 states
const i80States = ['CA', 'NV', 'UT', 'WY', 'NE', 'IA', 'IL', 'IN', 'OH', 'PA',
'NJ'];
```

```
for (const state of i80States) {
  // Configure event sources
  await configureEventSource({
    source_id: `${state.toLowerCase()}-i80`,
    url: `https://corridor.${state.toLowerCase()}.gov/api/tmc/events/tmdd?
corridor=I-80`,
    type: 'tmdd',
    poll_interval: 60
  });
}

// 2. Fleet operator subscribes to I-80 alerts
await axios.post('https://corridor.national.gov/api/notifications/subscribe', {
  subscriber_id: 'fleet-operator-123',
  subscription_type: 'webhook',
  delivery_target: 'https://api.fleet-company.com/corridor-alerts',
  filters: {
    corridors: ['I-80'],
    event_types: ['crash', 'road-closed', 'weather', 'construction'],
    min_severity: 'moderate'
  }
});

// 3. When Nebraska crash occurs, fleet gets immediate webhook
// Fleet system automatically:
// - Alerts drivers currently on I-80 in Nebraska
// - Reroutes trucks not yet in affected area
// - Updates ETAs for deliveries
// - Notifies customers of potential delays
```

Benefits:

- National visibility of corridor conditions
- Proactive route optimization
- Reduced delays and fuel costs
- Improved customer service

Use Case 4: Weather Event Coordination

Scenario: Blizzard warning issued for I-80 across multiple states.

Implementation:

```
// 1. National Weather Service issues CAP alert
const capAlert = `
<alert xmlns="urn:oasis:names:tc:emergency:cap:1.2">
  <info>
    <event>Blizzard Warning</event>
    <severity>Severe</severity>
    <area>
      <polygon>41.5,-104.0 41.5,-95.0 40.5,-95.0 40.5,-104.0 41.5,-104.0</polygon>
    </area>
```

```
  </info>
</alert>
`;

// 2. Corridor Communicator ingests CAP alert
await axios.post('https://corridor.national.gov/api/events/ingest', {
  source_id: 'nws',
  format: 'cap',
  data: capAlert
});

// 3. System determines I-80 impact across WY, NE, IA
const impactedStates = calculateImpact(capAlert.polygon, 'I-80');
// Result: ['WY', 'NE', 'IA']

// 4. Trigger coordinated response
for (const state of impactedStates) {
  // Post VMS messages
  await postVMSMessages(state, 'I-80', 'BLIZZARD WARNING - TRAVEL NOT ADVISED');

  // Send CAP alerts to 511 systems
  await send511Alert(state, event);

  // Notify TMC operators
  await notifyTMC(state, event);

  // Alert snow plow coordinators
  await notifySnowPlows(state, event);
}

// 5. Multi-state coordination call initiated
await notifyEmergencyCoordinators({
  event: 'Multi-State Blizzard',
  states: ['WY', 'NE', 'IA'],
  priority: 'critical'
});
```

Result: Coordinated response across state lines, consistent traveler messaging, proactive resource deployment

---

Security & Authentication

API Authentication Methods

1. API Key Authentication

For TMC/511 System Integration:

```
// Generate API key for external system
const apiKey = crypto.randomBytes(32).toString('hex');
const apiKeyHash = crypto.createHash('sha256').update(apiKey).digest('hex');
```

```
// Store in database
await db.run(`
  INSERT INTO integration_endpoints
  (endpoint_id, system_type, api_key_hash)
  VALUES (?, ?, ?)
`, ['nebraska-tmc', 'tmc', apiKeyHash]);

// External system includes API key in requests
const response = await axios.get('https://corridor.iowa.gov/api/tmc/events/tmdd', {
  headers: {
    'X-API-Key': apiKey
  }
});
```

Validation:

```
app.use('/api/tmc/*', (req, res, next) => {
  const providedKey = req.headers['x-api-key'];
  if (!providedKey) {
    return res.status(401).json({ error: 'API key required' });
  }

  const keyHash = crypto.createHash('sha256').update(providedKey).digest('hex');

  const endpoint = db.db.prepare(`
    SELECT * FROM integration_endpoints
    WHERE api_key_hash = ? AND enabled = true
  `).get(keyHash);

  if (!endpoint) {
    return res.status(401).json({ error: 'Invalid API key' });
  }

  req.endpoint = endpoint;
  next();
});
```

2. OAuth 2.0 (For 511 Mobile Apps)

```
const { AuthorizationCode } = require('simple-oauth2');

const oauth2 = new AuthorizationCode({
  client: {
    id: process.env.OAUTH_CLIENT_ID,
    secret: process.env.OAUTH_CLIENT_SECRET
  },
  auth: {
    tokenHost: 'https://auth.corridor.gov',
    tokenPath: '/oauth/token',
```

```
    authorizePath: '/oauth/authorize'
  }
});

// Token endpoint
app.post('/oauth/token', async (req, res) => {
  try {
    const result = await oauth2.getToken({
      code: req.body.code,
      redirect_uri: req.body.redirect_uri
    });

    res.json(result.token);
  } catch (error) {
    res.status(401).json({ error: 'Authentication failed' });
  }
});

// Protected endpoint
app.get('/api/511/events/json', authenticateOAuth, (req, res) => {
  // Return events
});
```

3. Webhook Signature Verification

Prevent tampering with webhook payloads:

```
function generateSignature(payload, secret) {
  const hmac = crypto.createHmac('sha256', secret);
  hmac.update(JSON.stringify(payload));
  return `sha256=${hmac.digest('hex')}`;
}

function verifySignature(payload, signature, secret) {
  const expectedSignature = generateSignature(payload, secret);
  return crypto.timingSafeEqual(
    Buffer.from(signature),
    Buffer.from(expectedSignature)
  );
}

// Send webhook with signature
const payload = { event_id: 'IA-I80-2025-001', ... };
const signature = generateSignature(payload, webhookSecret);

await axios.post(webhookUrl, payload, {
  headers: {
    'X-Corridor-Signature': signature,
    'Content-Type': 'application/json'
  }
});
```

```
// Receive and verify webhook
app.post('/webhooks/corridor-events', (req, res) => {
  const signature = req.headers['x-corridor-signature'];
  const payload = req.body;

  if (!verifySignature(payload, signature, webhookSecret)) {
    return res.status(401).json({ error: 'Invalid signature' });
  }

  // Process webhook
  processEvent(payload);
  res.json({ received: true });
});
```

IP Whitelisting

Restrict access to known TMC IP addresses:

```
const allowedIPs = [
  '192.168.1.0/24',     // Iowa DOT TMC
  '10.0.0.0/8',         // Nebraska DOT Network
  '172.16.0.0/12'       // Internal systems
];

function isIPAllowed(ip) {
  // Use ip-range-check library
  return allowedIPs.some(range => ipRangeCheck(ip, range));
}

app.use('/api/tmc/*', (req, res, next) => {
  const clientIP = req.ip || req.connection.remoteAddress;

  if (!isIPAllowed(clientIP)) {
    console.warn(`Blocked request from unauthorized IP: ${clientIP}`);
    return res.status(403).json({ error: 'Access denied' });
  }

  next();
});
```

Rate Limiting

Prevent abuse and ensure fair usage:

```
const rateLimit = require('express-rate-limit');

// TMC feed endpoints - higher limits
const tmcLimiter = rateLimit({
  windowMs: 1 * 60 * 1000, // 1 minute
  max: 120, // 120 requests per minute (1 every 500ms)
  message: 'Too many requests from this IP'
```

```
});

// 511 public endpoints - standard limits
const publicLimiter = rateLimit({
  windowMs: 1 * 60 * 1000,
  max: 60, // 60 requests per minute
  message: 'Rate limit exceeded'
});

app.use('/api/tmc/*', tmcLimiter);
app.use('/api/511/*', publicLimiter);
```

Encryption

TLS/SSL Required:

```
// Force HTTPS in production
if (process.env.NODE_ENV === 'production') {
  app.use((req, res, next) => {
    if (!req.secure) {
      return res.redirect(`https://${req.headers.host}${req.url}`);
    }
    next();
  });
}

// HSTS Header
app.use((req, res, next) => {
  res.setHeader('Strict-Transport-Security', 'max-age=31536000; includeSubDomains');
  next();
});
```

Monitoring & Alerting

System Health Monitoring

Monitor event ingestion pipeline:

```
const prometheus = require('prom-client');

// Metrics
const eventsIngested = new prometheus.Counter({
  name: 'events_ingested_total',
  help: 'Total number of events ingested',
  labelNames: ['source_id', 'event_type']
});

const eventProcessingDuration = new prometheus.Histogram({
  name: 'event_processing_duration_seconds',
  help: 'Event processing duration in seconds'
```

```
});

const notificationsSent = new prometheus.Counter({
  name: 'notifications_sent_total',
  help: 'Total notifications sent',
  labelNames: ['type', 'status']
});

const sourceErrors = new prometheus.Counter({
  name: 'source_errors_total',
  help: 'Total errors by source',
  labelNames: ['source_id']
});

// Instrument code
async function processEvent(event, sourceId) {
  const timer = eventProcessingDuration.startTimer();

  try {
    // Process event
    await storeEvent(event);
    eventsIngested.inc({ source_id: sourceId, event_type: event.type });
  } catch (error) {
    sourceErrors.inc({ source_id: sourceId });
    throw error;
  } finally {
    timer();
  }
}

async function sendNotification(type, target, event) {
  try {
    await deliverNotification(type, target, event);
    notificationsSent.inc({ type, status: 'success' });
  } catch (error) {
    notificationsSent.inc({ type, status: 'failed' });
    throw error;
  }
}

// Metrics endpoint
app.get('/metrics', async (req, res) => {
  res.set('Content-Type', prometheus.register.contentType);
  res.end(await prometheus.register.metrics());
});
```

Uptime Monitoring

Monitor external data sources:

```
const cron = require('node-cron');

// Check source health every 5 minutes
cron.schedule('*/5 * * * *', async () => {
  const sources = db.db.prepare(`
    SELECT * FROM event_sources WHERE enabled = true
  `).all();

  for (const source of sources) {
    try {
      const response = await axios.get(source.url, {
        timeout: 10000,
        headers: buildAuthHeaders(source)
      });

      // Update last success
      await db.run(`
        UPDATE event_sources
        SET last_success = ?, error_count = 0
        WHERE source_id = ?
      `, [new Date(), source.source_id]);

    } catch (error) {
      console.error(`Health check failed for ${source.name}:`, error.message);

      // Increment error count
      await db.run(`
        UPDATE event_sources
        SET error_count = error_count + 1
        WHERE source_id = ?
      `, [source.source_id]);

      // Alert if source down for extended period
      const errorCount = db.db.prepare(`
        SELECT error_count FROM event_sources WHERE source_id = ?
      `).get(source.source_id).error_count;

      if (errorCount >= 6) { // 30 minutes of failures
        await sendAlert({
          severity: 'critical',
          message: `Event source "${source.name}" has been down for 30+ minutes`,
          source_id: source.source_id
        });
      }
    }
  }
});
```

Logging

Comprehensive logging for troubleshooting:

```
const winston = require('winston');

const logger = winston.createLogger({
  level: 'info',
  format: winston.format.combine(
    winston.format.timestamp(),
    winston.format.json()
  ),
  transports: [
    new winston.transports.File({ filename: 'logs/error.log', level: 'error' }),
    new winston.transports.File({ filename: 'logs/combined.log' }),
    new winston.transports.Console({
      format: winston.format.simple()
    })
  ]
});

// Log event ingestion
logger.info('Event ingested', {
  event_id: event.event_id,
  source_id: source.source_id,
  event_type: event.event_type,
  severity: event.severity,
  corridor: event.corridor
});

// Log notifications
logger.info('Notification sent', {
  event_id: event.event_id,
  subscription_id: subscription.id,
  type: subscription.subscription_type,
  target: subscription.delivery_target,
  status: 'success'
});

// Log errors
logger.error('Event processing failed', {
  event_id: event.event_id,
  error: error.message,
  stack: error.stack
});
```

Dashboard

Real-time monitoring dashboard:

```
// WebSocket for live dashboard updates
const dashboardWSS = new WebSocket.Server({ port: 8081 });

dashboardWSS.on('connection', (ws) => {
  // Send current stats on connect
```

```
  ws.send(JSON.stringify({
    type: 'stats',
    data: getCurrentStats()
  }));
});

// Broadcast stats every 10 seconds
setInterval(() => {
  const stats = getCurrentStats();

  dashboardWSS.clients.forEach((client) => {
    if (client.readyState === WebSocket.OPEN) {
      client.send(JSON.stringify({
        type: 'stats',
        data: stats
      }));
    }
  });
}, 10000);

function getCurrentStats() {
  const activeEvents = db.db.prepare(`
    SELECT COUNT(*) as count FROM national_events WHERE status = 'active'
  `).get().count;

  const eventsByType = db.db.prepare(`
    SELECT event_type, COUNT(*) as count
    FROM national_events
    WHERE status = 'active'
    GROUP BY event_type
  `).all();

  const notificationsLast Hour = db.db.prepare(`
    SELECT COUNT(*) as count FROM notification_log
    WHERE sent_at > datetime('now', '-1 hour')
  `).get().count;

  return {
    active_events: activeEvents,
    events_by_type: eventsByType,
    notifications_sent_hour: notificationsLastHour,
    sources_active: getActiveSourceCount(),
    timestamp: new Date()
  };
}
```

Conclusion

This TMC & 511 System Integration Guide provides a comprehensive framework for connecting the DOT Corridor Communicator with Traffic Management Centers and 511 traveler information systems nationwide.

Key Capabilities Enabled

✅ **National Event Visibility** - Aggregate incidents from all 50 states ✅ **Real-Time Coordination** - Instant event sharing between TMCs ✅ **Traveler Information** - Automatic updates to 511 systems ✅ **Auto-Notifications** - Rule-based alerts to affected stakeholders ✅ **Multi-State Corridors** - Seamless coordination across state lines ✅ **Standards-Based** - TMDD, CAP, EDXL-DE, WZDx compliant

Implementation Checklist

Phase 1: Foundation (Weeks 1-2)

- [ ] Set up database schema
- [ ] Install dependencies
- [ ] Configure authentication
- [ ] Implement core API endpoints

Phase 2: TMC Integration (Weeks 3-4)

- [ ] TMDD XML feed endpoint
- [ ] WebSocket streaming
- [ ] VMS message posting
- [ ] Map layer GeoJSON

Phase 3: 511 Integration (Weeks 5-6)

- [ ] XML/JSON event feeds
- [ ] CAP alert distribution
- [ ] Webhook subscriptions
- [ ] IVR system integration

Phase 4: National Aggregation (Weeks 7-8)

- [ ] Event source configuration
- [ ] Polling workers
- [ ] Deduplication logic
- [ ] Event enrichment

Phase 5: Auto-Notifications (Weeks 9-10)

- [ ] Trigger rules engine
- [ ] Email/SMS delivery
- [ ] Push notifications
- [ ] Notification logging

Phase 6: Monitoring & Production (Weeks 11-12)

- [ ] Metrics & monitoring
- [ ] Health checks
- [ ] Logging infrastructure
- [ ] Load testing
- [ ] Security audit
- [ ] Documentation

- [ ] Training

Resources

Standards Documentation:

- TMDD: https://www.standards.its.dot.gov/Catalog/TMDD
- CAP: https://docs.oasis-open.org/emergency/cap/v1.2/
- EDXL-DE: https://www.oasis-open.org/committees/emergency/
- WZDx: https://github.com/usdot-jpo-ode/wzdx
- SAE J2354: https://www.sae.org/standards/content/j2354/

Related Documentation:

- JSTAN Integration Guide: `docs/JSTAN_INTEGRATION_GUIDE.md`
- JSTAN Quick Reference: `JSTAN_QUICK_REFERENCE.md`
- Grant Proposal Analyzer: `GRANT_PROPOSAL_ANALYZER_DOCUMENTATION.md`
- Connected Corridors Grants: `CONNECTED_CORRIDORS_GRANTS_INTEGRATION.md`

Support & Questions:

- For technical issues: Contact your system administrator
- Documentation: Available in the app under "📚 Docs"
- AASHTO JSTAN Committee: jstan@aashto.org

Version: 1.0 **Last Updated:** December 27, 2025 **Document:** TMC & 511 System Integration Guide
Component: DOT Corridor Communicator - National Event Feeds & Auto-Notifications