Real-Time Data Processing in Space Exploration

Ferenc Lengyel

May, 2024

Abstract

Develop real-time algorithms that can process data from satellites and spacecraft sensors as it is received. This could help in quicker decision-making for mission control and automated spacecraft adjustments. For more details, visit the project website: `https://spaceml.org/`.

Project Structure for Enhanced Real-Time Data Processing

1. **Project Setup**
   (a) 1.1 Environment Setup
   (b) 1.2 Dependencies
2. **Data Acquisition**
   (a) 2.1 API Integration
   (b) 2.2 Data Fetching
3. **Real-Time Processing**
   (a) 3.1 Data Filtering
   (b) 3.2 Data Transformation
   (c) 3.3 Data Storage
4. **Decision-Making Algorithms**
   (a) 4.1 Anomaly Detection
   (b) 4.2 Automated Adjustments
5. **User Interface**
   (a) 5.1 Dashboard Setup
   (b) 5.2 Data Visualization

6. **Testing and Deployment**
   (a) 6.1 Unit Testing
   (b) 6.2 Integration Testing
   (c) 6.3 Deployment

Project Setup

0.1 Environment Setup

Setting up the development environment for the Enhanced Real-Time Data Processing project involves the following steps:

1. **Install Python**:
   - Ensure Python 3.8+ is installed on your system.
   - Verify installation:

     ```
     python --version
     ```

2. **Create a Virtual Environment**:
   - Create and activate a virtual environment:

     ```
     python -m venv venv
     source venv/bin/activate # On Windows use `venv\Scripts\activate`
     ```

3. **Install Required Packages**:
   - Create a `requirements.txt` file with initial dependencies:

     ```
     touch requirements.txt
     ```

   - Add basic dependencies (e.g., `websockets`, `numpy`, `pandas`):

     ```
     websockets
     numpy
     pandas
     ```

   - Install dependencies:

     ```
     pip install -r requirements.txt
     ```

4. **Set Up Version Control**:
   - Initialize a Git repository and create a `.gitignore` file:

```
git init
touch .gitignore
echo "venv/" >> .gitignore
echo "__pycache__/" >> .gitignore
```

Example Commands:

```
# Verify Python installation
python --version

# Create and activate a virtual environment
python -m venv venv
source venv/bin/activate # On Windows use `venv\Scripts\activate`

# Create requirements.txt and install dependencies
touch requirements.txt
echo -e "websockets\nnumpy\npandas" > requirements.txt
pip install -r requirements.txt

# Initialize Git repository and set up .gitignore
git init
touch .gitignore
echo "venv/" >> .gitignore
echo "__pycache__/" >> .gitignore
```

Sources:

- Python Virtual Environments Documentation
- pip Documentation
- websockets Documentation
- numpy Documentation
- pandas Documentation

1 1.2 Dependencies

1.1 Dependencies

Setting up dependencies for the Enhanced Real-Time Data Processing project involves identifying and installing the necessary libraries and frameworks. This ensures our project has the required tools for data acquisition, real-time processing, and decision-making.

1. **Core Dependencies**:

- **websockets**: For real-time data acquisition via WebSockets.
- **numpy**: For numerical computations and data processing.
- **pandas**: For data manipulation and analysis.

2. **Visualization Libraries**:
   - **matplotlib**: For creating static, animated, and interactive visualizations.
   - **plotly**: For interactive graphing.
3. **Machine Learning**:
   - **scikit-learn**: For machine learning algorithms.
   - **tensorflow**: For deep learning models.
4. **Others**:
   - **flask**: For creating a web server to host the dashboard.
   - **requests**: For making HTTP requests.

Example `requirements.txt`:

```
websockets
numpy
pandas
matplotlib
plotly
scikit-learn
tensorflow
flask
requests
```

Installation:

```
pip install -r requirements.txt
```

Sources:

- websockets Documentation
- numpy Documentation
- pandas Documentation
- matplotlib Documentation
- plotly Documentation
- scikit-learn Documentation
- tensorflow Documentation
- flask Documentation
- requests Documentation

Data Acquisition

1.2 API Integration

Integrating APIs for acquiring data from satellites and spacecraft sensors is crucial for real-time processing. This involves:

1. **Identify API Endpoints**:
   - Determine the relevant APIs for satellite data (e.g., NASA, SpaceML).
2. **Authentication and Access**:
   - Set up authentication (API keys, OAuth) to access the APIs.
3. **Define Data Retrieval Functions**:
   - Write functions to call these APIs and handle responses.

Example Code:

```
import requests

# Load your API key from environment or configuration
API_KEY = 'your_api_key_here'
BASE_URL = 'https://api.spaceml.org/data'

def get_satellite_data(endpoint):
    url = f"{BASE_URL}/{endpoint}"
    headers = {'Authorization': f'Bearer {API_KEY}'}

    response = requests.get(url, headers=headers)

    if response.status_code == 200:
        return response.json()
    else:
        raise Exception(f"Error {response.status_code}: {response.text}")

# Example usage
try:
    data = get_satellite_data('satellite-endpoint')
    print(data)
except Exception as e:
    print(e)
```

Sources:

- NASA API Documentation
- SpaceML API Documentation

1.3 Data Fetching

Fetching data from APIs involves setting up a continuous process to collect and handle incoming data streams. Here's a structured approach:

1. **Polling vs. WebSockets**:
    - Decide between periodic polling of APIs or using WebSockets for real-time data streaming.
2. **Implement Polling**:
    - If using polling, set up scheduled tasks to periodically request data.
3. **Implement WebSocket Connection**:
    - For real-time streaming, establish a WebSocket connection to continuously receive data.

Example Code:

Polling

```
import time

def poll_satellite_data():
    while True:
        try:
            data = get_satellite_data('satellite-endpoint')
            process_data(data)
        except Exception as e:
            print(f"Error fetching data: {e}")
        time.sleep(60) # Poll every 60 seconds

poll_satellite_data()
```

WebSockets

```
import asyncio
import websockets

async def fetch_data():
    uri = "wss://api.spaceml.org/real-time"
    async with websockets.connect(uri) as websocket:
        while True:
            data = await websocket.recv()
            process_data(data)

asyncio.get_event_loop().run_until_complete(fetch_data())
```

Sources:

- websockets Documentation
- Python `time` module Documentation

Real-Time Processing

1.4 Data Filtering

Data filtering is essential to process and clean the incoming data for accurate analysis. This involves:

1. **Noise Reduction**:
   - Remove or smooth out noisy data.
2. **Outlier Detection**:
   - Identify and handle outliers that could skew results.
3. **Missing Data Handling**:
   - Fill or interpolate missing data points.
4. **Data Validation**:
   - Ensure data meets expected formats and ranges.

Example Code:

```
import numpy as np
import pandas as pd

def filter_data(data):
    # Convert to DataFrame for easier handling
    df = pd.DataFrame(data)

    # Noise reduction using a rolling mean
    df = df.rolling(window=3, min_periods=1).mean()

    # Detect and handle outliers (e.g., using z-score)
    df = df[(np.abs(df - df.mean()) <= (3 * df.std())).all(axis=1)]

    # Handle missing data (e.g., forward fill)
    df.fillna(method='ffill', inplace=True)

    # Validate data (e.g., ensuring within expected ranges)
    df = df[(df >= 0).all(axis=1)]

    return df.to_dict(orient='records')
```

```
# Example usage
data = [
    {'sensor1': 100, 'sensor2': 200},
    {'sensor1': 110, 'sensor2': None},
    {'sensor1': 95, 'sensor2': 210},
    {'sensor1': None, 'sensor2': 205}
]

filtered_data = filter_data(data)
print(filtered_data)
```

Sources:

- pandas Documentation
- numpy Documentation

1.5 Data Transformation

Data transformation is the process of converting raw data into a more suitable format for analysis. This involves:

1. **Normalization**:
    - Scale data to a standard range (e.g., 0 to 1).
2. **Aggregation**:
    - Summarize data points for specific intervals (e.g., hourly, daily).
3. **Feature Extraction**:
    - Derive new features from existing data (e.g., moving averages).

Example Code:

```
from sklearn.preprocessing import MinMaxScaler
import pandas as pd

def transform_data(data):
    df = pd.DataFrame(data)

    # Normalization
    scaler = MinMaxScaler()
    df[df.columns] = scaler.fit_transform(df[df.columns])

    # Aggregation (e.g., mean per hour)
    df['timestamp'] = pd.to_datetime(df['timestamp'])
```

```
    df.set_index('timestamp', inplace=True)
    df = df.resample('H').mean().reset_index()

    # Feature Extraction (e.g., rolling mean)
    df['rolling_mean'] = df['sensor1'].rolling(window=3).mean()

    return df.to_dict(orient='records')

# Example usage
data = [
    {'timestamp': '2024-05-18 10:00:00', 'sensor1': 100, 'sensor2': 200},
    {'timestamp': '2024-05-18 10:05:00', 'sensor1': 110, 'sensor2': 210},
    {'timestamp': '2024-05-18 10:10:00', 'sensor1': 95, 'sensor2': 205},
    {'timestamp': '2024-05-18 10:15:00', 'sensor1': 105, 'sensor2': 215}
]

transformed_data = transform_data(data)
print(transformed_data)
```

Sources:

- pandas Documentation
- scikit-learn Documentation

1.6 Data Storage

Data storage is critical for preserving transformed data for further analysis and historical reference. This involves:

1. **Choosing a Storage Solution**:
    - Decide between relational (e.g., PostgreSQL) and non-relational databases (e.g., MongoDB) based on data structure and access patterns.
2. **Setting Up the Database**:
    - Configure and connect to the database.
3. **Storing Data**:
    - Implement functions to save processed data into the chosen database.

Example Code:
Using PostgreSQL

```
import psycopg2
from psycopg2.extras import execute_values
```

```
def setup_database():
    conn = psycopg2.connect(
        dbname='satellite_data', user='user', password='password', host='localhost'
    )
    cursor = conn.cursor()
    cursor.execute("""
    CREATE TABLE IF NOT EXISTS data (
        id SERIAL PRIMARY KEY,
        timestamp TIMESTAMPTZ,
        sensor1 FLOAT,
        sensor2 FLOAT,
        rolling_mean FLOAT
    )
    """)
    conn.commit()
    return conn

def store_data(conn, data):
    cursor = conn.cursor()
    execute_values(
        cursor,
        "INSERT INTO data (timestamp, sensor1, sensor2, rolling_mean) VALUES %s",
        [(d['timestamp'], d['sensor1'], d['sensor2'], d['rolling_mean']) for d in data]
    )
    conn.commit()

# Example usage
data = [
    {'timestamp': '2024-05-18 10:00:00', 'sensor1': 100, 'sensor2': 200, 'rolling_mean': 102
    {'timestamp': '2024-05-18 10:05:00', 'sensor1': 110, 'sensor2': 210, 'rolling_mean': 105
    {'timestamp': '2024-05-18 10:10:00', 'sensor1': 95, 'sensor2': 205, 'rolling_mean': 103}
]

conn = setup_database()
store_data(conn, data)
conn.close()
```

Sources:

- psycopg2 Documentation
- PostgreSQL Documentation

Decision-Making Algorithms

1.7 Anomaly Detection

Anomaly detection involves identifying data points that deviate significantly from the norm. This is critical for detecting issues in real-time satellite and spacecraft data.

1. **Statistical Methods**:
   - Use statistical tests (e.g., z-score) to find anomalies.
2. **Machine Learning Models**:
   - Apply unsupervised learning algorithms (e.g., Isolation Forest, One-Class SVM).
3. **Threshold-Based Detection**:
   - Set predefined thresholds for sensor readings to flag anomalies.

Example Code:

```
from sklearn.ensemble import IsolationForest
import pandas as pd

def detect_anomalies(data):
    df = pd.DataFrame(data)

    # Using Isolation Forest for anomaly detection
    model = IsolationForest(contamination=0.05)
    df['anomaly'] = model.fit_predict(df[['sensor1', 'sensor2']])

    # Filter anomalies
    anomalies = df[df['anomaly'] == -1]

    return anomalies.to_dict(orient='records')

# Example usage
data = [
    {'sensor1': 100, 'sensor2': 200},
    {'sensor1': 110, 'sensor2': 210},
    {'sensor1': 95, 'sensor2': 205},
    {'sensor1': 500, 'sensor2': 600} # Outlier
]

anomalies = detect_anomalies(data)
print(anomalies)
```

Sources:

- scikit-learn IsolationForest Documentation
- Anomaly Detection in Python

1.8 Automated Adjustments

Automated adjustments involve responding to detected anomalies or specific conditions by adjusting spacecraft operations or mission parameters. This can include modifying control commands or updating system configurations.

1. **Define Adjustment Rules**:
   - Establish rules and conditions under which adjustments are triggered.
2. **Implement Adjustment Actions**:
   - Develop functions to execute specific adjustments (e.g., changing the orientation of solar panels).
3. **Integration with Decision-Making Algorithms**:
   - Integrate with anomaly detection to trigger adjustments automatically.

Example Code:

```
import requests

def send_adjustment_command(command):
    # Send command to spacecraft control system (simulated)
    response = requests.post('https://api.spaceml.org/control', json=command)
    if response.status_code == 200:
        print("Command sent successfully.")
    else:
        print(f"Failed to send command: {response.text}")

def automated_adjustments(anomalies):
    for anomaly in anomalies:
        if anomaly['sensor1'] > 400:
            command = {"action": "adjust_orientation", "parameters": {"angle": 15}}
            send_adjustment_command(command)
        elif anomaly['sensor2'] > 500:
            command = {"action": "reduce_power", "parameters": {"percentage": 20}}
            send_adjustment_command(command)

# Example usage
anomalies = [
```

```
    {'sensor1': 500, 'sensor2': 600},
    {'sensor1': 100, 'sensor2': 700}
]

automated_adjustments(anomalies)
```

Sources:

- HTTP Requests with Python - Requests Documentation
- Automated Control Systems Overview

User Interface

1.9 Dashboard Setup

Setting up a dashboard involves creating a web interface to visualize real-time data, anomalies, and automated adjustments. This allows users to monitor and control spacecraft operations effectively.

1. **Set Up Flask Server**:
    - Initialize a Flask application to serve the dashboard.
2. **Design Frontend**:
    - Use HTML, CSS, and JavaScript to create the user interface.
    - Integrate Plotly for interactive graphs and charts.
3. **Integrate Data Streams**:
    - Fetch and display real-time data on the dashboard.

Example Code:
Flask Setup

```
from flask import Flask, render_template, jsonify
import random

app = Flask(__name__)

@app.route('/')
def index():
    return render_template('index.html')

@app.route('/data')
def data():
    # Simulate real-time data
    data = {'sensor1': random.randint(90, 110), 'sensor2': random.randint(190, 210)}
```

```
    return jsonify(data)

if __name__ == '__main__':
    app.run(debug=True)
```

HTML (templates/index.html)

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Dashboard</title>
    <script src="https://cdn.plot.ly/plotly-latest.min.js"></script>
</head>
<body>
    <h1>Real-Time Dashboard</h1>
    <div id="chart"></div>
    <script>
        async function fetchData() {
            const response = await fetch('/data');
            const data = await response.json();
            Plotly.react('chart', [{y: [data.sensor1], type: 'line'}, {y: [data.sensor2], ty
        }
        setInterval(fetchData, 1000); // Update every second
    </script>
</body>
</html>
```

Sources:

- Flask Documentation
- Plotly Documentation

1.10 Data Visualization

Data visualization is crucial for interpreting and understanding real-time data. This subchapter focuses on implementing interactive visualizations using Plotly within a Flask application.

1. **Setup Plotly in Flask**:
    - Integrate Plotly to generate interactive charts.
2. **Update Visuals in Real-Time**:
    - Use JavaScript to fetch and update data dynamically.

Example Code:

Flask Endpoint for Data

```
from flask import Flask, render_template, jsonify
import random

app = Flask(__name__)

@app.route('/')
def index():
    return render_template('index.html')

@app.route('/data')
def data():
    data = {'sensor1': random.randint(90, 110), 'sensor2': random.randint(190, 210)}
    return jsonify(data)

if __name__ == '__main__':
    app.run(debug=True)
```

HTML with Plotly (templates/index.html)

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Data Visualization</title>
    <script src="https://cdn.plot.ly/plotly-latest.min.js"></script>
</head>
<body>
    <h1>Real-Time Data Visualization</h1>
    <div id="chart"></div>
    <script>
        async function fetchData() {
            const response = await fetch('/data');
            const data = await response.json();
            Plotly.react('chart', [
                { y: [data.sensor1], type: 'scatter', mode: 'lines', name: 'Sensor 1' },
                { y: [data.sensor2], type: 'scatter', mode: 'lines', name: 'Sensor 2' }
            ]);
        }
        setInterval(fetchData, 1000); // Update every second
    </script>
</body>
</html>
```

Sources:

- Flask Documentation
- Plotly Documentation

Testing and Deployment

1.11 Unit Testing

Unit testing is essential for ensuring the correctness of individual components within the real-time data processing application. This involves:

1. **Setting Up the Testing Framework**:
   - Use `unittest` or `pytest` for writing and running tests.
2. **Writing Test Cases**:
   - Create test cases for each function or module to verify their behavior.
3. **Running Tests**:
   - Execute the tests and ensure all cases pass.

Example Code:

```
import unittest
from data_processing import filter_data, transform_data

class TestDataProcessing(unittest.TestCase):

    def test_filter_data(self):
        data = [
            {'sensor1': 100, 'sensor2': 200},
            {'sensor1': 110, 'sensor2': None},
            {'sensor1': 95, 'sensor2': 210},
            {'sensor1': None, 'sensor2': 205}
        ]
        filtered = filter_data(data)
        self.assertEqual(len(filtered), 3) # Ensure outliers and missing data are handled

    def test_transform_data(self):
        data = [
            {'timestamp': '2024-05-18 10:00:00', 'sensor1': 100, 'sensor2': 200},
            {'timestamp': '2024-05-18 10:05:00', 'sensor1': 110, 'sensor2': 210},
            {'timestamp': '2024-05-18 10:10:00', 'sensor1': 95, 'sensor2': 205}
        ]
        transformed = transform_data(data)
        self.assertIn('rolling_mean', transformed[0]) # Check for the new feature
```

```
if __name__ == '__main__':
    unittest.main()
```

Sources:

- unittest Documentation
- pytest Documentation

1.12 Integration Testing

Integration testing ensures that different modules and components of the application work together as expected. This involves:

1. **Setup Test Environment**:
    - Create a separate configuration for testing.
2. **Write Integration Tests**:
    - Test interactions between components (e.g., data fetching, processing, and storage).
3. **Use Testing Framework**:
    - Utilize `pytest` for comprehensive test scenarios.

Example Code:

```
import pytest
from app import app, get_satellite_data, store_data, filter_data

@pytest.fixture
def client():
    app.config['TESTING'] = True
    with app.test_client() as client:
        yield client

def test_data_flow(client):
    # Simulate data fetching
    data = get_satellite_data('satellite-endpoint')
    assert data is not None

    # Process data
    filtered_data = filter_data(data)
    assert len(filtered_data) > 0

    # Store data
```

```
    conn = setup_database()
    store_data(conn, filtered_data)
    conn.close()

    # Fetch from database to verify
    cursor = conn.cursor()
    cursor.execute("SELECT * FROM data")
    result = cursor.fetchall()
    assert len(result) > 0

if __name__ == '__main__':
    pytest.main()
```

Sources:

- pytest Documentation
- Flask Testing

1.13 Deployment

Deploying the Enhanced Real-Time Data Processing application involves several steps:

1. **Choose a Hosting Service**:
    - Select a cloud platform (e.g., AWS, Heroku, Google Cloud) for hosting.
2. **Set Up the Server**:
    - Configure a virtual server or use Platform-as-a-Service (PaaS) for ease.
3. **Prepare the Application for Deployment**:
    - Ensure the application is production-ready (e.g., configure environment variables, database connections).
4. **Deploy the Application**:
    - Use deployment tools and scripts to deploy the application to the server.

Example Deployment Steps:
Using Heroku

1. **Install Heroku CLI**:
    - Follow instructions on Heroku CLI.

2. **Create a `Procfile`**:

   ```
   web: python app.py
   ```

3. **Initialize Git and Commit Code**:

   ```
   git init
   git add .
   git commit -m "Initial commit"
   ```

4. **Deploy to Heroku**:

   ```
   heroku create
   git push heroku master
   ```

5. **Scale the Application**:

   ```
   heroku ps:scale web=1
   ```

6. **Open the Application**:

   ```
   heroku open
   ```

Sources:

- Heroku Deployment Documentation
- AWS Elastic Beanstalk Documentation
- Google Cloud Platform Documentation