Ad Click Charging System

Pub-Sub Architecture Implementation

NIRAJ KUMAR BHANDARI
M25AI1063

Problem Statement

The Challenge

Online advertising platforms need to:

- Process millions of ad clicks daily
- Detect fraudulent clicks
- Charge advertisers accurately
- Track analytics in real-time
- Scale as traffic grows

Question: How do we build a system that can handle all this?

Solution - Pub-Sub Architecture

What is Pub-Sub?

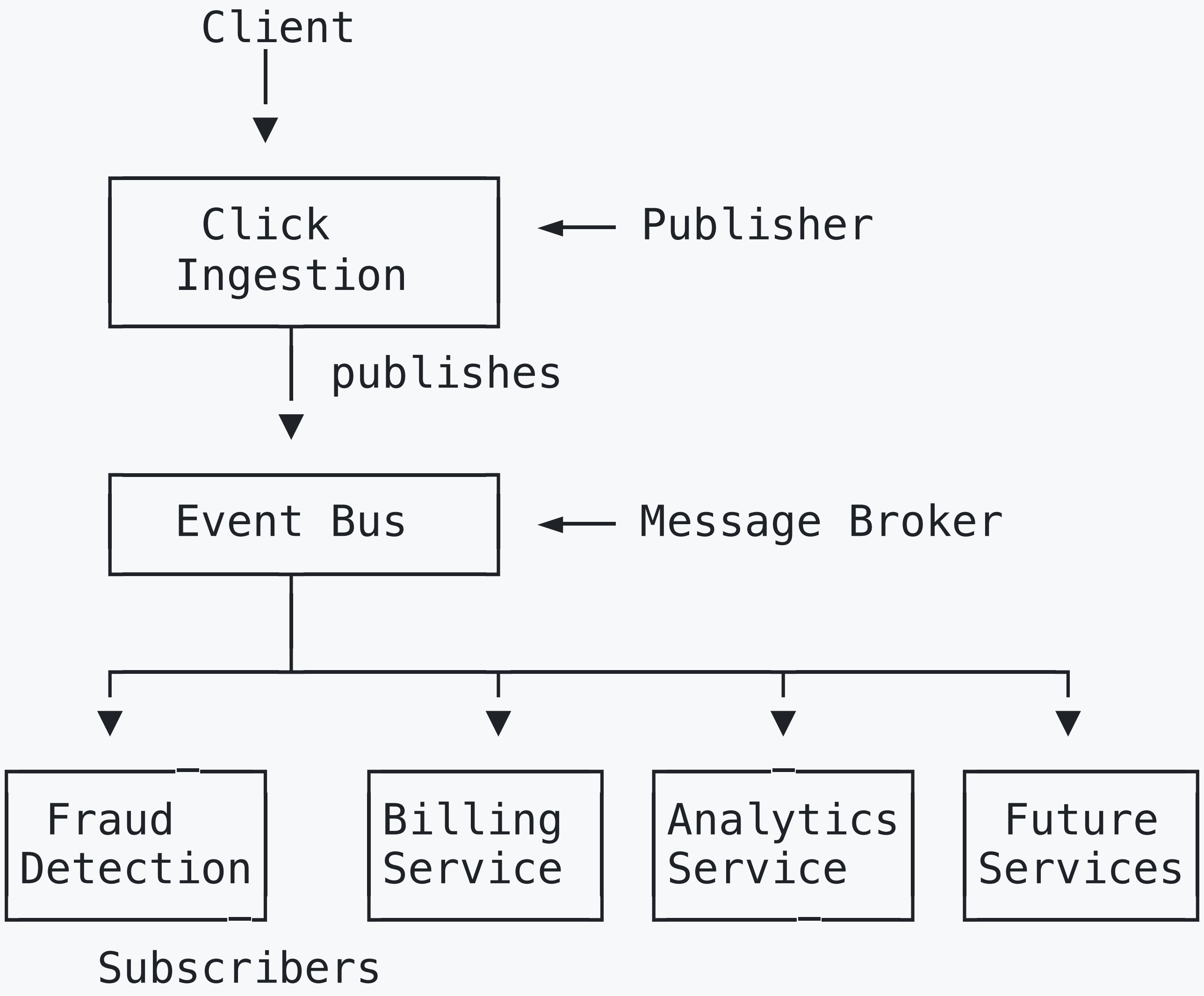
Publisher-Subscriber Pattern:

- Publishers send messages to topics
- Subscribers listen to topics they care about
- Publishers and subscribers don't know about each other

Benefits:

- Loose coupling
- Easy to scale
- Easy to add new features

System Architecture



Client

Click Ingestion

Publisher

publishes

Event Bus

Message Broker

Fraud Detection

Billing Service

Analytics Service

Future Services

Subscribers

Event Flow

How a Click is Processed

1. **Client** → Sends click via HTTP
2. **Click Ingestion** → Publishes to `click-events`
3. **Fraud Detection** → Checks click, publishes to `validated-clicks`
4. **Billing Service** → Charges advertiser, publishes to `billing-events`
5. **Analytics Service** → Updates metrics

Important to Note: Each service works independently.

Components

Publisher

Click Ingestion Service

- Receives HTTP requests
- Validates input
- Publishes events

Subscribers

Fraud Detection

- Analyzes clicks
- Calculates fraud score
- Blocks suspicious clicks

Fraud Detection

A simple scoring system to detect fraud

Scoring System:

- Missing user agent: +0.3
- Bot in user agent: +0.5
- Private IP address: +0.2
- Random factor: +0.0-0.2

Decision:

- Score >= 0.7 → Fraud (blocked)
- Score < 0.7 → Valid (processed)

Expected Outcome: It will protect advertisers from fraudulent charges.

Billing Logic

Calulation

```
Final Cost = Bid Amount × Quality Score × Time Adjustment
```

Quality Score = 1 - fraud_score

(Lower fraud = higher quality = charge more)

Time Adjustment:

- Peak hours (9am-5pm): ×1.2
- Off-peak hours: ×0.8

Budget Tracking:

- Each advertiser has budget
- Budget decreases with each charge

Demo - Budget Tracking

Real-Time Budget Updates

```
BILLING TRANSACTION
--------------------------------------------------
Advertiser: adv-501
Campaign: camp-101
Amount Charged: $0.60
Budget Before: $100.00
Budget After: $99.40
Total Spent: $0.60 / $100.00
Campaign Total: $0.60
--------------------------------------------------
```

We can observe the following:

- Amount charged
- Budget before/after

Quality Attribute #1 - Performance

Target: Fast Response Times

Goal: API response < 100ms

How We Achieved It:

- Asynchronous event processing
- In-memory event bus
- Minimal work per service

Results:

- Average response: ~50ms
- End-to-end processing: ~300ms

Important to Note: API returns immediately after publishing event

Quality Attribute #2 - Scalability

Target: Handle Growing Traffic

Goal: Support horizontal scaling

How We Achieved It:

- Stateless services
- Pub-sub decoupling
- Multiple subscribers per topic

Scaling Strategy:

```
Current: 1 publisher, 3 subscribers
Future:  3 publishers, 15 subscribers
         (5 fraud, 5 billing, 5 analytics)
```

Important to Note: Can run multiple instances of any service

Quality Attribute #3 - Reliability

Target: Handle Errors Gracefully

Goal: 99.9% uptime, no data loss

Key Actions:

- Budget validation (prevent overspending)
- Fraud detection (exclude bad clicks)
- Input validation (reject bad data)
- Error handling (graceful implementation)

Test Results:

- Budget exhaustion: Handled correctly
- Fraud detection: Working as expected

Demo

Demo Steps:

1. Start the system
2. Send test clicks
3. Watch the pub-sub flow
4. See budget tracking
5. View final statistics

Key Actions:

- Events flow through topics
- Fraud scores are calculated
- Budgets decrease
- Analytics get updated

Code Walkthrough

Key Code Snippets

Publishing an Event:

```
eventBus.publish('click-events', clickEvent);
```

Subscribing to an Event:

```
eventBus.subscribe('click-events', (clickEvent) => {
  this.checkForFraud(clickEvent);
});
```

Event Bus:

```
class EventBus extends EventEmitter {
  publish(topic, data) {
    this.emit(topic, data);
  }
```

Test Results

Test 1: Single Click

- Processed successfully
- All services triggered
- Budget updated

Test 2: Multiple Clicks (20)

- All processed correctly
- Budgets tracked accurately
- Campaign spending calculated

Test 3: Budget Exhaustion

- System prevents overspending
- Clicks rejected when budget

Production vs the Current System Implementation

What's Different?

Aspect	Current Implementation	Production
Message Broker	EventEmitter	Apache Kafka
Database	In-memory	MySQL
Fraud Detection	Rules	ML models
Scale	10s/sec	10,000s/sec
Deployment	1 machine	Cluster

Please note: Current Implementation uses same architecture, however scale is completely different.

Learning

Key Takeaways

1. **Pub-Sub is Powerful**
    - Decouples services
    - Enables scaling
    - Easy to extend
2. **Event-Driven Design**
    - Natural for async workflows
    - Good for high-volume systems
3. **Quality Attributes**
    - Must design them in

Future Enhancements

What's Next?

Short Term:

- Add MySQL database
- Implement Redis caching
- More fraud detection rules
- Web dashboard

Long Term:

- Apache Kafka integration
- Machine learning for fraud
- A/B testing
- Cloud deployment

Project Structure

Code Organization

```
src/
├── config/
│   └── event-bus.js            # Message broker
├── services/
│   ├── click-ingestion.js    # Publisher
│   ├── fraud-detection.js    # Subscriber
│   ├── billing-service.js    # Subscriber
│   └── analytics-service.js  # Subscriber
├── utils/
│   └── logger.js               # Logging
└── app.js                      # Main entry
```

Documentation

- **Architecture Diagrams**
- **Source Code** (GitHub)
- **README Files**
- **Project Report**
- **Quality Attributes Analysis**
- **Project Presentation (current presentation)**
- **Demo Recording**