A Project Report on the pub-sub implementation for an Ad Click Charging System

Overview

As part of this project, we have implemeted a pub-sub pattern to build an ad click charging system. The system uses multiple independent services, async communication, messaging queues and events to achieve highly scalable system.

Key Callouts:

- Implemented 4 services for the pub-sub model
- Implemented real-time budget tracking and fraud detection

1. Introduction

1.1 Problem Statement

Increasingly number of online platforms are employing some form of advertisement services for monetization. As these platform grow they need a scalable and efficient system to process millions of clicks per day. Such an system must:

- Handle high volume of clicks
- Detect fraudulent clicks
- Process billing accurately
- Track analytics in real-time/near real-time
- Scale seamlessly as traffic grows

1.2 Solution

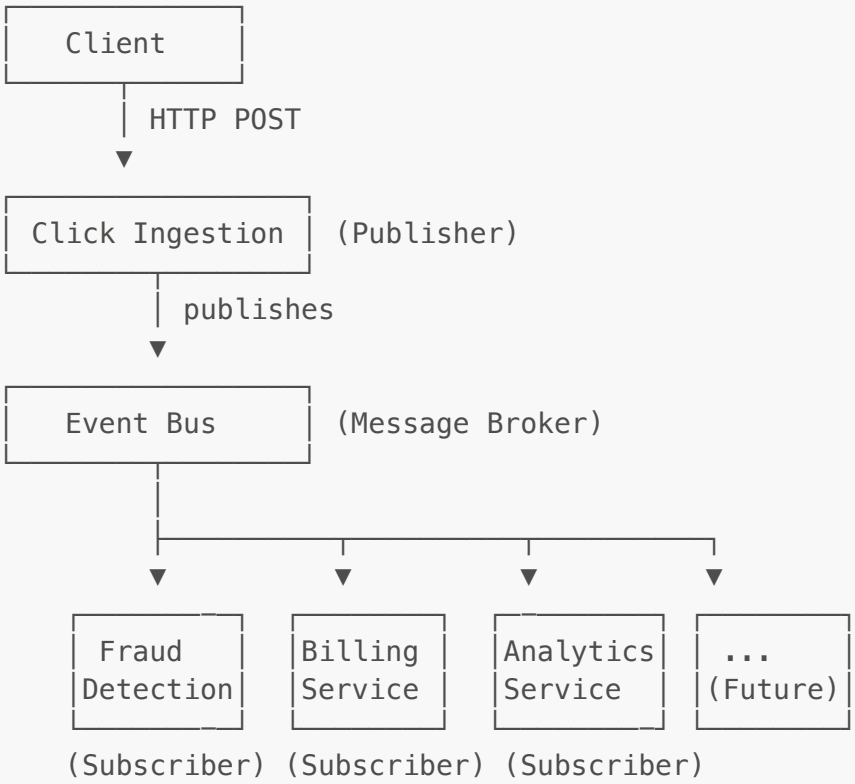
The choice of pub-sub model was driven by LinkedIn's implementation of events. The pub-sub makes it easy to add new services without modifyin existing ones, as the services don't need to know about other services. Pub-sub is also a good fit for event driven workflows and is scalable.

1.3 Learning Objectives

The Learning objectives are to understand pub-sub model, leverage this to implement an scalable event-drivem architecture. As a requirement of the project, we also need to understand and design for atleast three quality attributes.

2. System Architecture

2.1 High-Level Design



2.2 Components

Publisher:

- Click Ingestion Service: Receives HTTP requests and publishes events

Message Broker:

- Event Bus: Routes messages between publishers and subscribers

Subscribers:

- Fraud Detection: Validates clicks
- Billing Service: Charges advertisers
- Analytics Service: Tracks metrics

2.3 Event Flow

1. Client sends click via HTTP POST
2. Click Ingestion validates and publishes to `click-events`
3. Fraud Detection analyzes and publishes to `validated-clicks` or `fraud-alerts`
4. Billing Service processes payment and publishes to `billing-events`
5. Analytics Service updates metrics

---

3. Implementation Details

3.1 Project Structure

```
src/
├── config/
│   └── event-bus.js          # Pub-sub message broker
├── services/
│   ├── click-ingestion.js    # Publisher
│   ├── fraud-detection.js    # Subscriber 1
│   ├── billing-service.js    # Subscriber 2
│   └── analytics-service.js  # Subscriber 3
├── utils/
│   └── logger.js             # Logging utility
└── app.js                    # Main entry point
```

3.2 Fraud Detection Logic

Simple scoring algorithm:

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

If score >= 0.7, mark as fraud.

3.3 Billing Logic

Cost calculation:

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

Where:
- Quality Score = 1 - fraud_score
- Time Adjustment = 1.2 (peak hours) or 0.8 (off-peak)
```

Budget tracking:

- Each advertiser has initial budget
- Budget decreases with each charge
- Clicks rejected when budget exhausted

---

4. Quality Attributes

4.1 Performance

Target: API response < 100ms

Implementation:

- Asynchronous event processing

- In-memory event bus
- Minimal processing per service

Outcomes: Average response time ~50ms

4.2 Scalability

Target: Support horizontal scaling

Implementation:

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

Outcomes: Architecture supports N instances of each service

4.3 Reliability

Target: Handle errors gracefully

Implementation:

- Budget validation
- Fraud detection
- Input validation
- Error handling

Outcomes: System prevents overspending and fraud

Detailed Analysis: See `docs/architecture/quality-attributes-analysis.md`

---

5. Testing and Demonstration

5.1 Test Scenarios

Test 1: Single Click

```
curl -X POST http://localhost:3000/click \
  -H "Content-Type: application/json" \
  -d '{"ad_id":"ad-001","campaign_id":"camp-101","advertiser_id":"adv-
501","bid_amount":0.75}'
```

Result: Click processed through all services successfully

Test 2: Multiple Clicks

- Sent 20 clicks with varying parameters
- All processed correctly
- Budgets decreased appropriately

- Campaign spending tracked

Test 3: Budget Exhaustion

- Advertiser starts with $100 budget
- After spending $100, next click rejected
- System prevents overspending

5.2 Demo Output

The demo shows:

- Click received by ingestion
- Fraud score calculated
- Billing transaction details:
  - Amount charged
  - Budget before/after
  - Total spent
  - Campaign totals
- Analytics updated

Example output:

```
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
--------------------------------------------------
```

6. Comparison with Production Systems

6.1 Current Implementation vs Real Systems

Aspect	Current System	Production System
Message Broker	EventEmitter	Apache Kafka
Database	In-memory	MySQL/PostgreSQL
Fraud Detection	Rule-based	ML models
Scale	10s of clicks/sec	10,000s of clicks/sec
Deployment	Single machine	Distributed cluster

6.2 Learning outcomes

- Pub-sub pattern is powerful for decoupling
- Event-driven systems are naturally scalable
- Quality attributes require intentional design
- Production systems need more robust infrastructure

---

7. Conclusion

7.1 Project Outcomes

- Successfully implemented pub-sub architecture
- Demonstrated 3 quality attributes
- Created working demo with budget tracking

7.2 Key Learnings

1. **Pub-Sub Pattern**: Enables loose coupling and scalability
2. **Event-Driven Architecture**: Natural fit for asynchronous workflows
3. **Quality Attributes**: Must be designed in, not added later
4. **Simplicity**: Start simple, add complexity as needed