Pub/Sub Communications Architecture

EMQX-Based Messaging Infrastructure for Service Function Architecture

---

Executive Summary

The pub/sub communications system serves as the central nervous system for the Service Function Architecture platform. Built on EMQX MQTT, it provides orthogonal messaging capabilities that separate concerns cleanly: message routing, session management, security, and delivery guarantees are independent, composable components.

Core Design Principles:

- **Orthogonal Components**: Session management, topic routing, security, and reliability operate independently
- **Native EMQX Features**: Leverage built-in capabilities rather than building custom solutions
- **Simple Topic Hierarchy**: Predictable, flat topic structure without complex routing rules
- **Automatic Resource Management**: Native session expiry and cleanup without custom logic

Key Benefits:

- Sub-millisecond message routing with predictable performance
- Automatic session cleanup prevents resource leaks
- Fine-grained security without complex custom authorization layers
- Horizontal scaling through standard EMQX clustering
- Orthogonal design enables independent evolution of each component

---

Architecture Philosophy

Orthogonal Design Principles

The communications architecture follows Rich Hickey's principle of designing orthogonal components that can be composed without interference. Each subsystem operates independently:

Message Routing is concerned only with getting messages from publishers to subscribers efficiently, without knowledge of security, sessions, or delivery guarantees.

Session Management handles client lifecycle, expiry, and cleanup, without needing to understand message content or routing rules.

Security provides topic-level access control that operates independently of routing logic or session state.

Delivery Guarantees ensure message reliability through MQTT QoS levels, orthogonal to who sends messages or where they route.

This separation means changes to session management don't affect routing performance, security policy changes don't require message system modifications, and reliability improvements don't impact session handling.

Simplicity Over Ease

The system prioritizes simple, understandable designs over easy-to-use abstractions that hide complexity:

- **Topic names are explicit and predictable** rather than using complex routing rules
- **EMQX native features** are used directly rather than building custom abstractions
- **Security is declarative** through topic patterns rather than procedural authorization code
- **Session expiry leverages EMQX TTL** rather than custom cleanup mechanisms

---

EMQX Infrastructure Architecture

Cluster Design

High-Availability Configuration: The EMQX cluster runs as a 3-node deployment with automatic leader election and distributed session storage. Each node can handle the full message load, with automatic failover and session migration.

Performance Characteristics:

- Sub-millisecond message routing latency
- Millions of concurrent connections per cluster
- Linear scaling through node addition
- Automatic load balancing across nodes

Network Topology:

- HAProxy load balancer for connection distribution
- Direct TCP/WebSocket connections (no additional protocol layers)
- TLS termination at EMQX nodes for end-to-end encryption
- Management API for cluster health monitoring

Built-in Features Utilized

Session Persistence: EMQX native session storage with configurable TTL eliminates need for custom session management.

Message Routing: Native MQTT topic subscription with wildcard support provides all necessary routing capabilities.

Connection Management: Built-in connection pooling, keep-alive handling, and graceful disconnect processing.

Clustering: Distributed session state and automatic node discovery without external coordination services.

Topic Architecture

Hierarchical Topic Structure

The topic hierarchy is designed for clarity and efficient routing, with each level serving a specific architectural purpose:

```
service/{service-name}          # Service function invocation
session/{session-id}/response    # Client-specific responses
session/{session-id}/notifications  # Client notifications
ui/change/{user-id}           # User-specific UI updates
ui/change/group/{group-id}     # Group UI updates
system/{cluster-id}/errors      # System-wide error broadcasting
system/{cluster-id}/health      # Cluster health updates
```

Topic Usage Patterns

Service Function Communication:

- **Publishers:** Client applications, other service functions, UI components
- **Subscribers:** Service function pods listening for their specific service name
- **Message Flow:**
  - Client publishes `{"function": "createUser", "parameters": {...}}` to `service/UserManagement`
  - Service function processes request and publishes response to client's reply-to topic
  - Handles all business logic invocations across the platform

Client Session Management:

- **Publishers:** Service functions (responses), platform notification services
- **Subscribers:** Individual client applications using their session ID
- **Message Content:**
  - `session/{session-id}/response` - Function call responses, query results, operation confirmations
  - `session/{session-id}/notifications` - System alerts, user messages, background job completions
  - Each client only receives messages intended specifically for their session

User Interface Synchronization:

- **Publishers:** Service functions when data changes, UI components for collaborative features
- **Subscribers:** UI applications showing relevant data to users
- **Change Types:**
  - `ui/change/{user-id}` - Personal data updates (profile changes, personal dashboards, user preferences)
  - `ui/change/group/{group-id}` - Shared data visible to group members (team documents, project updates, shared calendars)
  - `ui/change/global` - Platform-wide changes (system announcements, feature releases, maintenance notices)

System Operations:

- **Publishers:** Platform infrastructure services, monitoring systems, operations tools
- **Subscribers:** Dashboards, monitoring tools, alerting systems, administrative interfaces
- **Event Types:**
  - `system/{cluster-id}/errors` - Infrastructure failures, service outages, performance degradation
  - `system/{cluster-id}/health` - Cluster status, node availability, performance metrics, capacity alerts

Topic Design Principles

Flat Hierarchy: No deep nesting or complex routing rules. Each topic level has a clear semantic meaning.

Predictable Patterns: Topic names follow consistent conventions that are immediately understandable.

Efficient Routing: Topic structure aligns with EMQX's native routing algorithms for optimal performance.

Security Alignment: Topic patterns map directly to security permissions without translation layers.

Message Flow Patterns

Service Function Calls:

1. Client publishes to `service/{service-name}` with reply-to topic
2. Service function processes and publishes response to reply-to topic
3. Client receives response on its session-specific topic

UI Synchronization:

1. Data changes trigger messages to `ui/change/{user-id}` or `ui/change/group/{group-id}`
2. Clients subscribe to their user/group topics
3. UI updates occur in real-time without polling

System Broadcasting:

1. Infrastructure components publish to `system/{cluster-id}/{message-type}`
2. Monitoring systems subscribe to system topics
3. Alerts and status updates propagate automatically

Message Envelope Structure

Standard Message Format: All messages follow a consistent envelope structure with headers and body, similar to HTTP but optimized for pub/sub messaging:

```json
{
  "messageId": "uuid-12345-67890",
  "correlationId": "corr-uuid-abcdef",
  "replyTo": "session/client-session-123/response",
  "headers": {
    "authorization": "Bearer eyJhbGciOiJSUzI1NiIs...",
    "content-type": "application/json",
    "user-agent": "Platform-Client/1.2.0",
    "x-tenant-id": "tenant-456",
    "x-request-id": "req-789",
    "x-timestamp": "2024-12-15T10:30:00Z",
    "x-message-ttl": "300"
  },
  "body": {
    "function": "createUser",
    "parameters": {
      "email": "user@example.com",
      "name": "John Doe"
    }
  }
}
```

Header Fields:

Authentication Headers:

- `authorization` - Bearer token (OIDC JWT) for user authentication
- `x-api-key` - Service-to-service authentication key
- `x-client-cert` - Client certificate fingerprint for mutual TLS

Routing Headers:

- `replyTo` - Topic where responses should be sent
- `correlationId` - Links requests with responses for async operations
- `x-tenant-id` - Multi-tenant isolation identifier

Metadata Headers:

- `content-type` - Message body format (application/json, application/octet-stream)
- `content-encoding` - Compression format (gzip, deflate)
- `user-agent` - Client identification string
- `x-timestamp` - Message creation timestamp
- `x-message-ttl` - Time-to-live in seconds
- `x-request-id` - Unique request identifier for tracing

Service Headers:

- `x-service-version` - Required service version for compatibility
- `x-retry-count` - Number of retry attempts made
- `x-priority` - Message priority level (high, normal, low)

Security Headers:

- `x-signature` - Message signature for integrity verification
- `x-encryption-key-id` - Key identifier for encrypted message bodies
- `x-access-token` - Short-lived access token for specific operations

Authentication Integration

OIDC Token Validation: Service functions automatically validate bearer tokens from message headers:

1. **Extract token** from `authorization` header
2. **Validate signature** against OIDC provider public keys
3. **Check expiration** and token validity
4. **Extract user context** (user ID, roles, permissions)
5. **Apply authorization** based on topic and operation

Multi-Tenant Security: The `x-tenant-id` header enables tenant isolation:

- Messages are automatically scoped to the specified tenant
- Cross-tenant access requires explicit permissions
- Service functions receive tenant context for data filtering

Service-to-Service Authentication: Internal service communication uses `x-api-key` headers:

- Each service has unique API keys for inter-service calls
- Keys are rotated automatically on schedule
- Failed authentication triggers security alerts

Message Processing Pipeline

Inbound Message Processing:

1. **Header Validation** - Check required headers and formats
2. **Authentication** - Validate tokens and extract user context
3. **Authorization** - Verify topic access permissions
4. **Deduplication** - Check message ID against recent cache
5. **Routing** - Forward to appropriate service function
6. **Processing** - Execute business logic with user context

Outbound Message Publishing:

1. **Envelope Creation** - Add standard headers and metadata
2. **Authentication Injection** - Include service credentials
3. **Encryption** - Apply message-level encryption if required
4. **Compression** - Compress large payloads
5. **QoS Application** - Set delivery guarantees
6. **Topic Publishing** - Send through EMQX broker

Error Response Format

Standard Error Envelope:

```json
{
 "messageId": "uuid-response-12345",
 "correlationId": "corr-uuid-abcdef",
 "headers": {
  "content-type": "application/json",
  "x-error-type": "validation_error",
  "x-timestamp": "2024-12-15T10:30:05Z"
 },
 "error": {
  "code": -32001,
  "message": "Validation Error",
  "details": {
   "field": "email",
   "constraint": "format",
   "value": "invalid-email"
  }
 }
}
```

Error Header Extensions:

- `x-error-type` - Classification of error (validation, business, system)
- `x-retry-after` - Suggested retry delay for transient errors
- `x-error-trace-id` - Correlation ID for error tracking
- `x-support-reference` - Reference number for customer support

Session Management

Native EMQX Session Expiry

Session Lifecycle: Sessions are managed entirely through EMQX's built-in session TTL mechanism. When clients connect, they specify a session expiry interval. EMQX automatically:

- Maintains session state for disconnected clients until expiry
- Queues messages for offline clients within session lifetime
- Automatically cleans up expired sessions and associated resources
- Fires session lifecycle events for platform notification

Session Configuration:

- Default session expiry: 30 minutes
- Maximum session expiry: 24 hours
- Cleanup interval: 5 minutes
- Session takeover enabled for reconnections

Session Security and Isolation

Session ID Generation: Cryptographically secure session IDs prevent session hijacking and ensure unique namespace separation.

Topic Isolation: Each session gets its own topic namespace (`session/{session-id}/*`) that other clients cannot access without explicit permission.

Automatic Cleanup: When sessions expire, EMQX automatically:

- Removes all session subscriptions
- Deletes queued messages
- Cleans up session metadata
- Triggers cleanup notifications

Session Event Integration

Platform Event Handling: EMQX publishes session lifecycle events to system topics:

- Client connected: `$SYS/brokers/+/clients/+/connected`
- Client disconnected: `$SYS/brokers/+/clients/+/disconnected`
- Session terminated: `$SYS/brokers/+/sessions/+/terminated`

Platform Response: Platform services subscribe to these system topics to:

- Update session tracking databases
- Trigger application-specific cleanup
- Send notifications about session state changes
- Update user presence information

Security Architecture

Topic-Level Authorization

Permission Model: Security operates at the topic level using pattern-based permissions. Users receive permissions for topic patterns rather than individual topics.

Authorization Patterns:

- `service/*` - Can invoke any service function
- `service/UserManagement` - Can only invoke UserManagement service
- `session/{user-session-id}/*` - Can access own session topics
- `ui/change/{user-id}` - Can trigger UI updates for specific user

OIDC Integration: Authentication uses standard OIDC tokens. EMQX validates tokens through HTTP callbacks to the platform's authentication service, which:

- Validates OIDC token with identity provider
- Maps user identity to topic permissions
- Returns allow/deny decisions to EMQX

Multi-Tenant Isolation

Tenant Topic Namespacing: Multi-tenant deployments use topic prefixes for isolation:

- `tenant/{tenant-id}/service/{service-name}`
- `tenant/{tenant-id}/session/{session-id}/*`
- `tenant/{tenant-id}/ui/change/*`

Tenant Security Boundaries: Users can only publish/subscribe to topics within their authorized tenants. Cross-tenant access requires explicit permissions.

Message Reliability

MQTT QoS Levels

QoS 0 (At Most Once): Used for non-critical messages like UI updates and notifications. Messages are delivered once or not at all, with minimal overhead.

QoS 1 (At Least Once): Used for service function calls and important notifications. Messages are guaranteed delivery with possible duplicates.

QoS 2 (Exactly Once): Reserved for critical operations requiring exactly-once semantics, such as financial transactions or data mutations.

Deduplication Strategy

Message ID-Based Deduplication: Each message includes a unique message ID and correlation ID. Services maintain a bounded cache of recently processed message IDs to detect and ignore duplicates.

Time-Based Expiry: Messages include expiration timestamps. Expired messages are automatically discarded without processing.

Replay Protection: Correlation IDs prevent replay attacks by ensuring each unique operation can only be processed once within a time window.

Real-Time UI Communications

UI Update Architecture

Direct MQTT Integration: User interfaces connect directly to EMQX using WebSocket MQTT, eliminating intermediary layers and reducing latency. This direct connection enables true real-time updates without polling or complex state synchronization mechanisms.

UI Topic Patterns:

```
ui/change/{user-id}           # User-specific UI updates
ui/change/group/{group-id}      # Group-based UI updates
ui/change/global              # Platform-wide UI updates
ui/data/{entity-type}/{entity-id} # Entity-specific data changes
ui/form/{form-id}/validation    # Real-time form validation
ui/collaboration/{document-id}   # Collaborative editing updates
```

UI Topic Usage Details

Personal UI Updates (`ui/change/{user-id}`):

- **Publishers:** Service functions after processing user-specific operations
- **Subscribers:** User's active UI sessions (web browser, mobile app, desktop application)
- **Content Examples:**
  - Profile changes: `{"type": "profile_updated", "fields": ["avatar", "displayName"]}`
  - Preference updates: `{"type": "preferences_changed", "theme": "dark", "language": "en"}`
  - Personal dashboard data: `{"type": "dashboard_refresh", "widgets": ["calendar", "tasks"]}`
  - Notification state: `{"type": "notifications_read", "count": 0}`

Group UI Updates (`ui/change/group/{group-id}`):

- **Publishers:** Service functions handling group/team operations, collaborative services
- **Subscribers:** All group members' UI sessions currently viewing group-related content
- **Content Examples:**
  - Team document changes: `{"type": "document_updated", "documentId": "doc123", "author": "user456"}`
  - Project status updates: `{"type": "project_status", "projectId": "proj789", "status": "in_progress"}`
  - Shared calendar events: `{"type": "event_added", "eventId": "evt101", "title": "Team Meeting"}`
  - Group membership changes: `{"type": "member_added", "userId": "user999", "role": "editor"}`

Global UI Updates (`ui/change/global`):

- **Publishers:** System administration services, platform management tools
- **Subscribers:** All active UI sessions that should receive platform-wide updates
- **Content Examples:**
  - System announcements: `{"type": "announcement", "message": "Maintenance window scheduled", "priority": "high"}`
  - Feature releases: `{"type": "feature_enabled", "feature": "advanced_search", "allUsers": true}`
  - Emergency notifications: `{"type": "emergency", "message": "Service degradation detected", "action": "save_work"}`

Entity Data Updates (`ui/data/{entity-type}/{entity-id}`):

- **Publishers:** Service functions that modify specific entities (users, orders, products, etc.)
- **Subscribers:** UI components displaying that specific entity's data
- **Content Examples:**
  - User data: `ui/data/user/123` → `{"type": "user_updated", "changes": {"status": "active", "lastLogin": "2024-12-15T10:30:00Z"}}`
  - Order status: `ui/data/order/456` → `{"type": "order_status_changed", "status": "shipped", "trackingNumber": "1Z999"}`
  - Product inventory: `ui/data/product/789` → `{"type": "inventory_updated", "quantity": 45, "reserved": 12}`

Form Validation (`ui/form/{form-id}/validation`):

- **Publishers:** Validation services, business rule engines
- **Subscribers:** Active form instances requiring real-time validation feedback
- **Content Examples:**
  - Field validation: `{"field": "email", "valid": false, "message": "Email already exists"}`
  - Cross-field validation: `{"fields": ["startDate", "endDate"], "valid": true}`
  - Business rule validation: `{"rule": "credit_limit", "valid": false, "message": "Exceeds available credit"}`

Collaborative Editing (`ui/collaboration/{document-id}`):

- **Publishers:** Users actively editing the document, collaboration services
- **Subscribers:** All users currently viewing or editing the same document
- **Content Examples:**
  - Text operations: `{"type": "text_insert", "position": 245, "text": "Hello", "author": "user123"}`
  - Cursor positions: `{"type": "cursor_move", "user": "user456", "position": 300, "selection": [300, 305]}`
  - User presence: `{"type": "user_joined", "user": "user789", "cursor": {"position": 0, "color": "#FF5733"}}`
  - Conflict resolution: `{"type": "operation_transform", "original": {...}, "transformed": {...}}`

Two-Way Data Binding with Server Objects

Server-Side State Objects: Service functions maintain stateful objects that are automatically synchronized with UI components. Changes to server objects trigger immediate UI updates through the messaging system.

Binding Mechanism:

- UI components bind to server object properties
- Server object changes publish to user-specific UI topics
- UI receives updates and automatically re-renders affected components
- User interactions send updates back to server objects through service calls

State Change Propagation: When a service function modifies an object that affects UI state:

1. **Determine affected users/groups** based on object ownership and permissions
2. **Calculate delta changes** to minimize message payload size
3. **Publish to relevant UI topics** with change details and new state
4. **UI frameworks receive and apply updates** automatically to bound components

Conflict Resolution: For collaborative scenarios, updates include version vectors or timestamps to resolve conflicts automatically. The last-writer-wins for simple cases, with merge strategies for complex collaborative objects.

Live Data Synchronization

Entity Change Propagation: When service functions modify entities, they publish change notifications to relevant UI topics:

1. **User-Specific Changes:** Updates that affect a single user's view
2. **Group Changes:** Updates visible to multiple users in a group/organization
3. **Global Changes:** System-wide updates affecting all users
4. **Entity Changes:** Specific object modifications with fine-grained targeting

Subscription Management: UI clients automatically subscribe to relevant topics based on:

- Current user identity and permissions
- Active page/component requirements
- Group memberships and roles
- Real-time collaboration contexts

Collaborative Features

Multi-User Editing: Real-time collaborative editing uses operational transformation over MQTT:

- Document changes publish to `ui/collaboration/{document-id}`
- All collaborators receive updates immediately
- Operational transforms resolve concurrent edits
- Presence information shows active collaborators

Live Cursors and Awareness: User presence and cursor positions broadcast to collaboration topics, enabling:

- Live cursor positions in shared documents
- User awareness indicators (who's viewing what)
- Typing indicators and activity status
- Real-time comment and discussion threads

Conflict-Free Updates: For non-collaborative scenarios, updates use simple last-writer-wins with optimistic UI updates:

- UI immediately reflects user changes
- Server validates and confirms or rejects changes
- Conflicts trigger UI notifications and potential rollbacks

UI Performance Optimization

Selective Updates: UI frameworks receive only relevant change notifications:

- Component-level subscriptions to specific data elements
- Automatic subscription cleanup when components unmount
- Batched updates for multiple rapid changes
- Delta updates containing only changed fields

Connection Management: UI applications maintain persistent MQTT connections with:

- Automatic reconnection on network issues
- Session restoration after disconnects
- Queued message delivery for offline periods
- Graceful degradation during connectivity problems

Bandwidth Optimization:

- JSON patch format for minimal update payloads
- Message compression for large updates
- Update coalescing for rapid state changes
- Priority-based delivery for critical vs cosmetic updates

Form and Validation Integration

Real-Time Validation: Forms connect to validation services through MQTT for immediate feedback:

- Field-level validation on blur/change events
- Cross-field validation for dependent fields
- Server-side business rule validation
- Asynchronous validation for external data checks

Form State Synchronization: Multi-step forms maintain state across sessions:

- Form progress automatically saved to server
- Users can resume forms from any device
- Draft state preserved during browser crashes
- Collaborative form completion for team processes

UI Framework Integration

Framework Agnostic Design: The messaging system works with any UI framework through standardized adapters:

- React hooks for MQTT subscriptions and server state binding
- Vue.js composables for reactive server object integration
- Angular services for real-time data synchronization
- Web Components for framework-independent reusable UI elements

Declarative Binding Syntax: UI templates use declarative syntax to bind to server objects:

```javascript
// React example
const { userData, updateUser } = useServerObject('user', userId);

// Vue example
const userData = useServerObject('user', userId);

// Angular example
userData$ = this.serverObjects.observe('user', userId);
```

State Management Integration: For applications using state management libraries:

- Redux/Vuex stores automatically sync with server state
- MobX observables mirror server object changes
- Context providers manage subscription lifecycles
- State hydration from server on initial load

Error Handling and Offline Support

Graceful Degradation: UI applications handle messaging failures gracefully:

- Cached data displayed during connectivity issues
- Queued actions replayed when connection restored
- User notifications about offline state
- Optimistic updates with rollback on errors

Conflict Resolution: When offline changes conflict with server state:

- Automatic merge strategies for simple conflicts
- User-driven conflict resolution for complex cases
- Version history for understanding change sequences
- Rollback capabilities for problematic updates

Progressive Enhancement: Applications work without real-time features and enhance when messaging is available:

- Basic functionality through traditional HTTP requests
- Real-time enhancements when MQTT connection succeeds
- Automatic feature detection and adaptation
- Consistent user experience across connection qualities

Performance and Scaling

Horizontal Scaling

EMQX Cluster Scaling:

- Add nodes to cluster for increased capacity
- Automatic session migration during scaling
- Load balancer updates for new nodes
- No application changes required

Topic Sharding: For extremely high-volume topics, messages can be distributed across multiple topic shards using consistent hashing.

Connection Distribution: HAProxy distributes connections across cluster nodes based on connection count and node health.

Performance Monitoring

Key Metrics:

- Message throughput per topic
- Connection count per node
- Session count and expiry rates
- Topic subscription counts
- Message delivery latency

Alerting Thresholds:

- Node memory usage >90%
- Message queue depth >10,000
- Connection count >80% of limit
- Cluster nodes down
- High message delivery failures

Operational Considerations

High Availability

Cluster Resilience:

- 3-node cluster survives single node failure
- Automatic failover for crashed nodes
- Session migration during node maintenance
- Load balancer health checks prevent routing to failed nodes

Data Persistence:

- Session data persisted across cluster
- Message queuing for offline clients
- Automatic backup and recovery
- Cross-data center replication support

Monitoring and Observability

Built-in Metrics: EMQX provides comprehensive metrics through its management API and system topics:

- Connection and session statistics
- Message delivery rates and failures
- Topic subscription counts
- Node resource utilization

Platform Integration: Platform monitoring services subscribe to EMQX system topics to:

- Track application-level message flows
- Correlate messaging metrics with business operations
- Generate alerts for message delivery failures
- Analyze communication patterns

Designed for Change

Orthogonal Components: Each subsystem can evolve independently without affecting others:

- Topic structure changes don't impact session management
- Security policy updates don't require routing modifications
- Performance optimizations don't affect message reliability
- New message patterns don't disrupt existing flows

EMQX Feature Adoption: As EMQX adds new capabilities, they can be adopted incrementally:

- Shared subscriptions for load balancing
- Message transformation and routing rules
- Enhanced security features
- Performance optimizations

Platform Evolution: The messaging architecture supports platform growth:

- New service types use existing topic patterns
- Additional message reliability levels through QoS
- Enhanced security through finer-grained topic permissions
- Improved observability through additional system topics

Simplicity Preservation

Resistance to Complexity: The architecture actively resists complexity accumulation:

- New features must fit existing orthogonal design
- Custom solutions avoided when EMQX provides native capabilities
- Topic hierarchy remains flat and predictable
- Security model stays declarative and pattern-based

This ensures the system remains understandable and maintainable as it scales and evolves.

Conclusion

The pub/sub communications architecture provides a simple, orthogonal foundation for platform messaging. By leveraging EMQX's native capabilities and maintaining clear separation of concerns, the system delivers high performance and reliability while remaining comprehensible and maintainable.

The design prioritizes simplicity over ease, resulting in a system where each component can be understood and modified independently. This architectural approach supports both current requirements and future evolution without accumulating complexity.