

The Publish–Subscribe Architectural Style

A Comprehensive Reference for Event-Driven Distributed Systems

Contents

1	Overview	2
1.1	Scope and Applicability	2
1.2	Historical Context	2
1.3	Relationship to Other Styles	3
2	Elements	3
2.1	Publisher and Subscriber Components	3
2.1.1	Types of Publishing Components	3
2.1.2	Types of Subscribing Components	3
2.1.3	Essential Properties of Components	4
2.2	Publish–Subscribe Connectors	4
2.2.1	Types of Publish–Subscribe Connectors	4
2.2.2	Essential Properties of Connectors	5
2.3	Events	5
2.3.1	Event Structure	5
2.3.2	Event Design Considerations	6
3	Relations	6
3.1	Attachment Relation	6
3.1.1	Properties of Attachment	6
3.2	Publishes Relation	6
3.2.1	Properties of Publishes	7
3.3	Subscribes-To Relation	7
3.3.1	Properties of Subscribes-To	7
3.4	Event Flow Relation	7
4	Computational Model	7
4.1	Basic Event Distribution	7
4.2	Subscription Management	7
4.3	Event Delivery Models	8
4.4	Synchronous vs. Asynchronous Publication	8
4.5	Event Ordering	8
4.6	Concurrent Event Processing	9
5	Constraints	9

5.1	Connector Constraints	9
5.2	Visibility Constraints	9
5.3	Self-Subscription Constraints	9
5.4	Topology Constraints	9
5.5	Quality of Service Constraints	10
5.6	Component Role Constraints	10
6	What the Style is For	10
6.1	Decoupling Event Producers from Consumers	10
6.2	Supporting GUI Frameworks	10
6.3	Enabling Notification Systems	11
6.4	Supporting Enterprise Integration	11
6.5	Enabling Event-Driven Architecture	11
6.6	Supporting Real-Time Data Distribution	11
7	Notations	11
7.1	Box-and-Line Diagrams	11
7.2	UML Component Diagrams	12
7.3	UML Sequence Diagrams	12
7.4	Event Flow Diagrams	12
7.5	Topic Hierarchies	12
7.6	Event Schemas	12
7.7	Architecture Description Languages	12
8	Quality Attributes	12
8.1	Performance	12
8.2	Scalability	13
8.3	Reliability	13
8.4	Availability	13
8.5	Security	13
8.6	Modifiability	14
8.7	Testability	14
9	Common Publish–Subscribe Patterns	14
9.1	Topic-Based Publish–Subscribe	14
9.2	Content-Based Publish–Subscribe	14
9.3	Fan-Out Pattern	14
9.4	Fan-In Pattern	15
9.5	Consumer Groups Pattern	15
9.6	Event Sourcing Pattern	15
9.7	Saga Pattern	15
9.8	Event-Carried State Transfer Pattern	15
9.9	Dead Letter Pattern	15
10	Implementation Considerations	16
10.1	Connector Implementation Options	16
10.2	Subscription Management	16
10.3	Event Storage	16

10.4	Serialization	17
10.5	Delivery Guarantees Implementation	17
10.6	Ordering Implementation	17
11	Examples	17
11.1	GUI Event Handling	17
11.2	Message Queue System	17
11.3	Event Streaming Platform	18
11.4	Cloud Event Services	18
11.5	Microservices Event Bus	18
11.6	Real-Time Updates	18
12	Best Practices	18
12.1	Design Events Carefully	19
12.2	Choose Appropriate Granularity	19
12.3	Handle Failures Explicitly	19
12.4	Manage Schema Evolution	19
12.5	Consider Ordering Requirements	19
12.6	Monitor Event Flow	20
12.7	Secure Event Infrastructure	20
12.8	Design for Scale	20
13	Common Challenges	20
13.1	Event Ordering	20
13.2	Exactly-Once Delivery	21
13.3	Subscriber Backpressure	21
13.4	Event Schema Evolution	21
13.5	Debugging and Tracing	21
13.6	Testing	22
13.7	Operations and Monitoring	22
14	Conclusion	22

1 Overview

The publish–subscribe style is a component-and-connector architectural style that structures a system around the production, distribution, and consumption of events. Publishers announce events without knowledge of which components will receive them; subscribers express interest in events without knowledge of which components produce them. An intermediary—the publish–subscribe connector or event bus—manages the distribution of events from publishers to interested subscribers.

This fundamental decoupling of event producers from event consumers is the defining characteristic of the style. Publishers and subscribers interact indirectly through the event distribution mechanism rather than directly with each other. This indirection enables loose coupling, dynamic reconfiguration, and scalable event distribution.

The publish–subscribe style underlies much of modern distributed systems architecture, from user interface frameworks to enterprise integration to real-time data streaming. Its ability to connect components without creating direct dependencies makes it invaluable for building flexible, extensible, and scalable systems.

1.1 Scope and Applicability

The publish–subscribe style applies to systems where components need to communicate through events without tight coupling. This includes graphical user interface frameworks where UI components react to user events and state changes, enterprise application integration where systems exchange business events across organizational boundaries, real-time data distribution where market data, sensor readings, or telemetry flow to multiple consumers, Internet of Things architectures where devices publish data consumed by various applications, microservices communication where services react to events from other services, notification systems where users or systems receive alerts about relevant events, collaborative applications where changes propagate to multiple participants, and monitoring and observability systems where metrics, logs, and traces flow to analysis tools.

The style is particularly valuable when event producers should not need to know about consumers, when the set of consumers may change dynamically, when events may need to reach multiple consumers, when producers and consumers may operate at different rates, and when system components should be independently deployable and evolvable.

1.2 Historical Context

The publish–subscribe pattern has deep roots in computing history. Early event systems in operating systems and GUI frameworks established the pattern of registering callbacks for events. The Observer pattern in object-oriented design formalized the concept for in-process communication. Message-oriented middleware like IBM MQ and TIBCO brought publish–subscribe to enterprise integration. The Java Message Service (JMS) standardized messaging APIs. Modern distributed streaming platforms like Apache Kafka have scaled publish–subscribe to massive throughput. Cloud-native event services from AWS, Azure, and Google provide managed publish–subscribe infrastructure.

Understanding this evolution helps architects recognize the breadth of publish–subscribe implementations and select appropriate technologies for their context.

1.3 Relationship to Other Styles

The publish–subscribe style relates to several other architectural patterns.

It contrasts with client–server where communication is direct and synchronous rather than indirect and typically asynchronous.

It relates to the pipe-and-filter style but differs in that events go to multiple subscribers rather than a single downstream filter, and the flow is determined by subscriptions rather than fixed connections.

It is often combined with microservices architecture for inter-service communication.

It can implement the event-driven architecture pattern where system behavior is determined by event occurrence and handling.

It supports the event sourcing pattern where state changes are captured as a sequence of events.

Many systems combine publish–subscribe with other styles. A microservices system might use client–server for synchronous queries and publish–subscribe for asynchronous event propagation.

2 Elements

The publish–subscribe style comprises components that publish or subscribe to events and connectors that distribute events from publishers to subscribers.

2.1 Publisher and Subscriber Components

Any component-and-connector component with at least one publish or subscribe port participates in the publish–subscribe style. Components may be purely publishers, purely subscribers, or both.

2.1.1 Types of Publishing Components

Publishing components generate events for distribution.

Event sources are components whose primary purpose is generating events, such as sensors, user interface controls, or system monitors.

State-change publishers announce events when their internal state changes, enabling other components to react to changes.

Command result publishers announce the outcomes of operations, enabling interested parties to learn about completed actions.

Aggregating publishers collect events from multiple sources and publish derived or consolidated events.

Bridge publishers receive events from external systems and republish them within the publish–subscribe network.

2.1.2 Types of Subscribing Components

Subscribing components receive and process events.

Event handlers receive events and perform actions in response, such as updating state, triggering operations, or producing side effects.

Event processors transform events, potentially publishing new derived events.

Event loggers persist events for audit, replay, or analysis purposes.

Event aggregators collect multiple events to compute summaries or detect patterns.

Bridge subscribers receive events and forward them to external systems.

User interface components update displays in response to events.

2.1.3 Essential Properties of Components

When documenting publish–subscribe components, architects should capture several property categories.

Event production properties describe what events are announced. Event types list the types of events the component may publish. Publication conditions describe when events are published. Event content describes the data included in published events. Publication rate describes how frequently events are published.

Event consumption properties describe what events are received. Subscribed events list the event types the component subscribes to. Subscription filters describe conditions that narrow received events. Processing semantics describe how received events are handled. Consumption rate describes how fast events can be processed.

Blocking properties describe synchronization behavior. Publisher blocking indicates conditions under which a publisher is blocked, such as when buffers are full or delivery is synchronous. Subscriber blocking indicates whether subscribers block event delivery.

Quality of service properties describe delivery requirements. Delivery guarantee specifies at-most-once, at-least-once, or exactly-once requirements. Ordering requirements specify whether event order must be preserved. Latency requirements specify acceptable delay from publication to delivery.

2.2 Publish–Subscribe Connectors

The publish–subscribe connector manages event distribution from publishers to subscribers. It has announce roles for publishers and listen roles for subscribers.

2.2.1 Types of Publish–Subscribe Connectors

Connectors vary in their distribution mechanisms and capabilities.

Direct notification connectors maintain subscriber lists and invoke subscribers directly when events occur. This is the simplest form, common in single-process observer patterns.

Message broker connectors use an intermediary broker that receives events from publishers, stores them temporarily, and delivers them to subscribers. This decouples publisher and subscriber availability.

Topic-based connectors organize events into named topics. Publishers send to topics; subscribers subscribe to topics. This is the most common form in message-oriented middleware.

Content-based connectors route events based on event content rather than topics. Subscribers specify predicates over event attributes. This provides fine-grained filtering but requires more complex routing.

Type-based connectors route events based on event type hierarchy. Subscribers can receive events of a type and all its subtypes.

Channel-based connectors provide point-to-point channels that can be configured for publish–subscribe through fan-out.

2.2.2 Essential Properties of Connectors

Distribution properties describe how events reach subscribers.

Delivery model specifies push (connector initiates delivery to subscribers), pull (subscribers request events), or hybrid approaches.

Fan-out describes how events reach multiple subscribers: unicast (one subscriber), multicast (group of subscribers), or broadcast (all subscribers).

Routing strategy describes how the connector determines which subscribers receive which events.

Persistence properties describe event storage.

Durability indicates whether events survive connector restart.

Retention specifies how long events are retained.

Replay indicates whether subscribers can receive past events.

Reliability properties describe delivery guarantees.

Delivery guarantee specifies at-most-once (events may be lost), at-least-once (events may be duplicated), or exactly-once (each event delivered once).

Ordering indicates whether events are delivered in publication order.

Acknowledgment describes how delivery confirmation is handled.

Performance properties characterize connector efficiency.

Throughput specifies the maximum event rate.

Latency describes delay from publication to delivery.

Scalability describes how capacity changes with publishers, subscribers, and event rate.

2.3 Events

Events are the data items flowing through the publish–subscribe system. While not architectural elements themselves, their structure significantly affects system design.

2.3.1 Event Structure

Events typically contain several categories of information.

Event metadata includes event type or topic, unique event identifier, timestamp of occurrence, source identifier, and correlation identifiers for related events.

Event payload contains the domain-specific data relevant to the event, structured according to a schema.

Event headers contain routing and processing information used by the connector.

2.3.2 Event Design Considerations

Event granularity ranges from fine-grained (many small events) to coarse-grained (fewer large events). Fine-grained events provide flexibility but increase overhead. Coarse-grained events reduce overhead but may include unnecessary data for some subscribers.

Event completeness describes whether events are self-contained (including all relevant data) or referential (including identifiers to look up data). Self-contained events reduce coupling but increase size.

Event schema management becomes important as systems evolve. Schema registries, versioning strategies, and compatibility rules help manage event evolution.

3 Relations

Relations in the publish–subscribe style define how components connect to the event distribution infrastructure.

3.1 Attachment Relation

The *attachment* relation associates components with the publish–subscribe connector by prescribing which components announce events and which components have registered to receive events.

3.1.1 Properties of Attachment

Role determines whether the attachment is for publishing (announce role) or subscribing (listen role). A component may have attachments of both types.

Event scope defines which events flow through the attachment. For publishers, this specifies which event types may be announced. For subscribers, this specifies the subscription filter determining which events are received.

Binding time indicates when the attachment is established. Static binding is fixed at design or deployment time. Dynamic binding allows attachments to change at runtime through subscription and unsubscription.

Durability indicates whether the attachment persists across component or connector restarts. Durable subscriptions receive events published while the subscriber was unavailable.

3.2 Publishes Relation

The *publishes* relation indicates that a component produces events of specific types. This relation captures the potential event flow from a component.

3.2.1 Properties of Publishes

Event types enumerate the types of events the component may publish.

Conditions describe when events are published.

Guarantees describe what the publisher commits to regarding event content and timing.

3.3 Subscribes-To Relation

The *subscribes-to* relation indicates that a component receives events matching a subscription. This relation captures the potential event flow to a component.

3.3.1 Properties of Subscribes-To

Subscription filter specifies which events are received, based on topic, type, content predicates, or combinations.

Handler specifies what happens when matching events arrive.

Quality of service specifies delivery requirements for this subscription.

3.4 Event Flow Relation

The *event flow* relation describes the runtime flow of events from publishers through the connector to subscribers. This emergent relation results from the combination of publications and subscriptions.

Understanding event flow is crucial for analyzing system behavior, performance, and failure modes.

4 Computational Model

The computational model describes how publish–subscribe systems execute.

4.1 Basic Event Distribution

The fundamental computation follows a predictable pattern. A component produces an event and announces it through its publish port. The publish–subscribe connector receives the event. The connector determines which subscribers should receive the event based on their subscriptions. The connector dispatches the event to each matching subscriber. Each subscriber’s listen port receives the event. The subscriber processes the event.

This basic model admits many variations in timing, ordering, and delivery guarantees.

4.2 Subscription Management

Subscriptions must be managed throughout system operation.

Subscription registration occurs when a component expresses interest in events. The subscription specifies selection criteria (topics, types, predicates) and may specify quality of service requirements.

Subscription storage maintains the set of active subscriptions for routing decisions. Storage may be centralized in the connector or distributed.

Subscription matching determines which subscriptions match a published event. Matching complexity depends on subscription expressiveness.

Subscription removal occurs when a component loses interest. Graceful unsubscription notifies the connector. Failure detection identifies subscriptions to dead components.

4.3 Event Delivery Models

Events can be delivered through different models.

Push delivery has the connector actively send events to subscribers. Subscribers must be ready to receive. This provides low latency but requires subscriber availability.

Pull delivery has subscribers request events from the connector. Subscribers control timing. This handles varying subscriber speeds but may increase latency.

Long polling has subscribers make requests that block until events are available. This provides push-like behavior over pull-based protocols.

Streaming has the connector maintain persistent connections for continuous event flow. This reduces connection overhead for high-frequency events.

4.4 Synchronous vs. Asynchronous Publication

Publication can be synchronous or asynchronous.

Synchronous publication blocks the publisher until events are delivered (or at least acknowledged by the connector). This provides delivery confirmation but couples publisher performance to subscriber and connector performance.

Asynchronous publication returns immediately after the connector accepts the event. This decouples publisher from delivery but provides weaker guarantees.

Fire-and-forget publication provides no confirmation at all. This provides maximum decoupling but no delivery guarantee.

4.5 Event Ordering

Ordering semantics vary across implementations.

Total ordering delivers all events to all subscribers in the same order. This simplifies reasoning but limits scalability.

Per-publisher ordering delivers events from each publisher in publication order, but events from different publishers may interleave differently at different subscribers.

Per-topic ordering maintains order within topics but not across topics.

Causal ordering preserves happens-before relationships but allows concurrent events to be delivered in any order.

No ordering guarantee allows arbitrary reordering, which provides maximum flexibility for the implementation.

4.6 Concurrent Event Processing

Subscribers may process events concurrently.

Serial processing handles one event at a time. This is simple but may limit throughput.

Parallel processing handles multiple events concurrently. This improves throughput but requires thread-safe handlers.

Partitioned processing assigns events to processing lanes based on a partition key. Events with the same key are processed serially; different keys are processed in parallel.

5 Constraints

The publish–subscribe style imposes constraints that define valid architectural configurations.

5.1 Connector Constraints

All components are connected to an event distributor that may be viewed as either a bus (connector) or a component. This central distribution point is fundamental to the style.

Publish ports are attached to announce roles. Subscribe ports are attached to listen roles. Port and role types must match.

5.2 Visibility Constraints

Constraints may restrict which components can listen to which events.

Topic-based restrictions limit subscriptions to specific topics based on component identity or role.

Content-based restrictions limit subscriptions based on event content criteria.

Security restrictions enforce access control over event visibility.

5.3 Self-Subscription Constraints

Constraints may specify whether a component can listen to its own events.

Allowing self-subscription enables components to react to their own publications, which can be useful for consistency.

Prohibiting self-subscription prevents potential infinite loops and simplifies reasoning.

5.4 Topology Constraints

Constraints may restrict the publish–subscribe topology.

Single connector constraints require all components connect to one event distributor, simplifying the architecture.

Multiple connector constraints allow multiple event distributors, potentially with bridging between them.

Hierarchical constraints organize connectors in hierarchies for scalability.

5.5 Quality of Service Constraints

Constraints may impose quality of service requirements.

Delivery guarantees may be mandated for certain event types.

Latency bounds may be required for time-sensitive events.

Ordering requirements may be imposed for certain event flows.

5.6 Component Role Constraints

A component may be both a publisher and a subscriber, by having ports of both types. However, constraints may restrict this.

Some designs require separation between event producers and consumers.

Some designs prohibit certain combinations of publications and subscriptions to prevent cycles or enforce layering.

6 What the Style is For

The publish–subscribe style supports several important architectural goals.

6.1 Decoupling Event Producers from Consumers

The primary benefit is sending events to unknown recipients, isolating event producers from event consumers.

Publishers need not know how many subscribers exist, who the subscribers are, or how subscribers will use events.

Subscribers need not know how many publishers exist, who the publishers are, or why publishers produce events.

This decoupling enables independent evolution—publishers and subscribers can change independently as long as event contracts are maintained. It enables dynamic topology since subscribers can come and go without publisher changes. It enables scaling since additional subscribers can be added without publisher modification. It reduces coordination since teams can work on publishers and subscribers independently.

6.2 Supporting GUI Frameworks

The style provides core functionality for graphical user interface frameworks.

User interface controls publish events when users interact with them.

Event handlers subscribe to control events to implement application behavior.

The Model-View-Controller and related patterns use publish–subscribe for model-to-view notification.

This enables separation between UI controls and application logic, reusable controls that work with any subscriber, and dynamic binding of controls to behavior.

6.3 Enabling Notification Systems

The style enables various notification systems including mailing lists where messages are published to lists and delivered to subscribers, bulletin boards where posts are visible to all subscribers, social networks where user activities generate events for followers, and alert systems where conditions trigger notifications to interested parties.

6.4 Supporting Enterprise Integration

The style is fundamental to enterprise application integration.

Business events from one system can be consumed by other systems.

Systems can be integrated without point-to-point connections.

New consumers can be added without modifying producers.

This enables loose coupling between enterprise systems, gradual system modernization through event-based integration, real-time data distribution across the enterprise, and audit and compliance through event logging.

6.5 Enabling Event-Driven Architecture

The style supports event-driven architectural patterns.

Event sourcing captures state changes as events.

Event-driven microservices communicate through published events.

Complex event processing detects patterns across event streams.

CQRS (Command Query Responsibility Segregation) uses events to synchronize read models.

6.6 Supporting Real-Time Data Distribution

The style enables real-time data distribution scenarios.

Market data distribution delivers prices to trading systems.

IoT data distribution delivers sensor readings to applications.

Telemetry distribution delivers metrics to monitoring systems.

Live updates deliver content changes to connected clients.

7 Notations

Publish–subscribe architectures can be represented using various notations.

7.1 Box-and-Line Diagrams

Informal diagrams show components, connectors, and event flows. Publishers and subscribers are shown as boxes. The event bus or broker is shown as a central element. Arrows show event flow direction. Labels identify event types or topics.

Conventions vary; some show the connector explicitly while others show it as a shared medium.

7.2 UML Component Diagrams

UML provides notation for publish–subscribe. Components represent publishers and subscribers. Interfaces represent event types. Dependencies show subscription relationships. The publish–subscribe connector may be shown as a component or elided.

7.3 UML Sequence Diagrams

Sequence diagrams show event flow over time. Lifelines represent publishers, subscribers, and the connector. Messages show event publication and delivery. Asynchronous messages (half arrowheads) represent typical pub–sub interaction.

7.4 Event Flow Diagrams

Specialized diagrams focus on event flow. Event sources are shown on the left. Event sinks are shown on the right. Event types or topics are shown as channels. Flow lines show which publishers produce and which subscribers consume each event type.

7.5 Topic Hierarchies

Topic-based systems may document topic structure. Tree diagrams show topic hierarchies. Wildcards indicate subscription patterns. Annotations describe topic semantics.

7.6 Event Schemas

Event structure is documented through schemas. JSON Schema, Avro, Protocol Buffers, or other schema languages define event structure. Schema registries provide centralized schema documentation. Version histories track event evolution.

7.7 Architecture Description Languages

Formal ADLs can specify publish–subscribe architectures. Event types are formally defined. Publication and subscription relations are explicit. Quality of service requirements can be specified.

8 Quality Attributes

Publish–subscribe decisions significantly affect system quality attributes.

8.1 Performance

Performance in publish–subscribe systems has distinctive characteristics.

Throughput depends on the connector’s ability to route events and the subscribers’ ability to process them. Fan-out to many subscribers multiplies the delivery work.

Latency accumulates through publication, routing, delivery, and processing. Asynchronous processing can reduce apparent latency for publishers.

Resource usage includes connector resources for routing and buffering, publisher resources for event creation, and subscriber resources for event processing.

Performance tactics include partitioning to distribute load, batching to amortize overhead, filtering to reduce delivered volume, and caching to speed routing decisions.

8.2 Scalability

Scalability in publish–subscribe addresses growth in multiple dimensions.

Publisher scalability handles increasing event sources through partitioning and load distribution.

Subscriber scalability handles increasing consumers through connector scaling and consumer groups.

Event rate scalability handles increasing event volume through horizontal scaling and streaming architectures.

Topic scalability handles increasing event types through efficient routing data structures.

8.3 Reliability

Reliability ensures events are delivered despite failures.

Delivery guarantees specify whether events may be lost (at-most-once), may be duplicated (at-least-once), or are delivered exactly once.

Persistence enables recovery after failures by storing events durably.

Acknowledgment confirms successful delivery, enabling retry of unacknowledged events.

Redundancy replicates connector infrastructure for fault tolerance.

8.4 Availability

Availability ensures the system accepts and delivers events.

Publisher availability depends on connector availability to accept events.

Subscriber availability is decoupled from publishers when connectors buffer events.

Connector availability is critical; redundancy and failover are essential for high availability.

Graceful degradation may continue accepting events when some subscribers are unavailable.

8.5 Security

Security in publish–subscribe addresses multiple concerns.

Authentication verifies publisher and subscriber identity.

Authorization controls who can publish to topics and who can subscribe.

Encryption protects event content in transit and at rest.

Audit logging records publications and subscriptions for compliance.

Input validation prevents malicious event content from causing harm.

8.6 Modifiability

The style strongly supports modifiability through decoupling.

Adding subscribers requires no publisher changes.

Adding publishers requires no subscriber changes.

Event schema evolution requires coordination but is localized.

Connector changes are transparent to well-designed publishers and subscribers.

8.7 Testability

The style supports testability through separation.

Publishers can be tested by verifying published events.

Subscribers can be tested by injecting test events.

Integration tests verify event flow through the system.

Event logging provides visibility for debugging.

9 Common Publish–Subscribe Patterns

Several recurring patterns address common publish–subscribe challenges.

9.1 Topic-Based Publish–Subscribe

Events are organized into named topics. Publishers send events to topics. Subscribers subscribe to topics by name. All events on a topic go to all topic subscribers.

This pattern is simple and widely supported. Topic hierarchies with wildcards provide flexibility. It is the dominant pattern in messaging middleware.

Considerations include topic naming conventions, topic granularity, and topic lifecycle management.

9.2 Content-Based Publish–Subscribe

Subscribers specify predicates over event content. Events are delivered to subscribers whose predicates match. No predefined topics are required.

This pattern provides fine-grained filtering without proliferating topics. It supports complex subscription criteria. However, it requires more sophisticated routing.

Considerations include predicate language expressiveness, routing efficiency, and subscription management complexity.

9.3 Fan-Out Pattern

One publisher’s events go to many subscribers. The connector replicates events to all matching subscribers. Subscribers process events independently.

This pattern supports broad notification, data distribution, and parallel processing. It is the fundamental publish–subscribe distribution model.

Considerations include delivery ordering across subscribers, handling slow subscribers, and managing subscriber failures.

9.4 Fan-In Pattern

Many publishers send events to common topics or channels. Events from multiple sources are aggregated. Subscribers see a unified event stream.

This pattern supports event aggregation, monitoring consolidation, and multi-source processing.

Considerations include event ordering across publishers, attribution of event sources, and handling high aggregate volume.

9.5 Consumer Groups Pattern

Multiple subscriber instances form a group. Each event goes to one instance in the group. Load is distributed across instances.

This pattern enables horizontal scaling of event processing, high availability through redundancy, and work distribution.

Considerations include partition assignment strategy, rebalancing on membership changes, and exactly-once processing.

9.6 Event Sourcing Pattern

All state changes are captured as events. Events are the source of truth. State is derived by replaying events.

This pattern enables complete audit trail, temporal queries, and system reconstruction. It requires careful event design and replay infrastructure.

9.7 Saga Pattern

Long-running transactions are decomposed into events. Each step publishes events triggering subsequent steps. Compensating events handle failures.

This pattern enables distributed transactions without distributed locks. It requires careful failure handling and idempotent operations.

9.8 Event-Carried State Transfer Pattern

Events carry complete state, not just change notifications. Subscribers maintain local state copies. No callback to publishers is needed.

This pattern reduces coupling and enables offline operation. It increases event size and requires consistency management.

9.9 Dead Letter Pattern

Undeliverable or unprocessable events go to a dead letter queue. Failed events are isolated for analysis. Processing continues for other events.

This pattern prevents poison messages from blocking processing. It enables debugging and manual intervention.

10 Implementation Considerations

Implementing publish–subscribe systems involves several practical considerations.

10.1 Connector Implementation Options

The publish–subscribe connector can be implemented in various ways.

In-process notification uses direct method calls or callbacks within a process. This is simple and fast but limited to single process.

Message brokers provide dedicated servers that receive, store, and deliver messages. Examples include RabbitMQ, ActiveMQ, and cloud services. This supports distributed systems with persistence and reliability.

Distributed logs provide append-only logs with consumer offsets. Apache Kafka exemplifies this approach. This supports high throughput and replay.

Multicast uses network-level multicast for efficient one-to-many delivery. This is efficient for local networks but limited in scope.

Peer-to-peer uses direct connections between publishers and subscribers with a discovery mechanism. This avoids central bottlenecks but adds complexity.

10.2 Subscription Management

Subscriptions must be managed throughout system operation.

Subscription storage may be centralized in the connector or distributed across nodes. Centralized storage simplifies consistency; distributed storage improves scalability.

Subscription indexing enables efficient matching of events to subscriptions. Data structures depend on subscription expressiveness.

Subscription durability determines whether subscriptions survive restarts. Durable subscriptions enable recovering missed events.

10.3 Event Storage

Event persistence enables reliability and replay.

In-memory storage provides fast access but loses events on failure.

Disk-based storage provides durability but adds latency.

Replicated storage provides fault tolerance through redundancy.

Tiered storage balances speed and capacity through hot and cold storage.

Retention policies determine how long events are kept. Time-based retention removes old events. Size-based retention removes events when capacity is reached. Compaction retains only the latest event per key.

10.4 Serialization

Events must be serialized for transmission and storage.

Text formats like JSON and XML provide human readability and flexibility. They have parsing overhead and verbosity.

Binary formats like Protocol Buffers, Avro, and MessagePack provide efficiency and schemas. They require tooling and are not human-readable.

Schema evolution must be considered. Compatible changes allow independent evolution. Breaking changes require coordinated updates.

10.5 Delivery Guarantees Implementation

Different delivery guarantees require different mechanisms.

At-most-once requires no special handling. Events are delivered if possible and forgotten.

At-least-once requires acknowledgment and retry. Events are redelivered until acknowledged.

Exactly-once requires deduplication or transactional delivery. Events are delivered once through idempotency or transactions.

10.6 Ordering Implementation

Ordering guarantees require coordination.

Total ordering requires single-threaded processing or consensus protocols.

Partition ordering uses consistent partitioning so related events go to the same partition, which is processed serially.

Logical ordering uses timestamps or sequence numbers for subscribers to reorder.

11 Examples

Concrete examples illustrate publish–subscribe concepts.

11.1 GUI Event Handling

Graphical user interfaces extensively use publish–subscribe.

User interface controls publish events for user interactions: button clicks, text changes, and selections. Application code subscribes to control events to implement behavior. The event loop dispatches events to registered handlers.

Modern frameworks like React use variations. State changes publish notifications. Components subscribe to state. The framework optimizes rendering based on subscriptions.

11.2 Message Queue System

Enterprise message queues implement publish–subscribe.

Producers send messages to topics or queues. Consumers subscribe to receive messages. The broker stores messages and manages delivery.

Example with RabbitMQ: Publishers send to exchanges. Exchanges route to queues based on bindings. Consumers read from queues. Acknowledgments confirm processing.

11.3 Event Streaming Platform

Streaming platforms implement publish–subscribe at scale.

Apache Kafka organizes events into topics. Topics are partitioned for parallelism. Producers append events to partitions. Consumers read from partitions, tracking their position. Consumer groups distribute partitions among instances.

This enables high-throughput event processing, event replay from any position, long-term event retention, and stream processing through event transformation.

11.4 Cloud Event Services

Cloud providers offer managed publish–subscribe.

AWS SNS and SQS provide topic-based pub-sub with queue-based delivery. Azure Event Grid routes events based on topics and filters. Google Cloud Pub/Sub provides global event distribution with exactly-once delivery.

These services handle infrastructure, scaling, and operations, enabling focus on event design and processing logic.

11.5 Microservices Event Bus

Microservices architectures use publish–subscribe for integration.

Services publish domain events when significant state changes occur. Other services subscribe to events they need to react to. An event bus or broker distributes events. Services remain loosely coupled through event contracts.

Example: An order service publishes OrderPlaced events. Inventory service subscribes to reserve stock. Shipping service subscribes to schedule delivery. Notification service subscribes to email confirmation.

11.6 Real-Time Updates

Web applications use publish–subscribe for real-time updates.

Servers publish events when data changes. WebSocket connections deliver events to browsers. Client-side code updates the user interface.

Example: A collaborative document editor publishes edit events. All connected clients subscribe. Changes appear in real-time across all sessions.

12 Best Practices

Experience suggests several best practices for publish–subscribe systems.

12.1 Design Events Carefully

Events are contracts between publishers and subscribers.

Include all necessary information to make events self-contained when appropriate.

Version events from the start to enable evolution.

Use meaningful names that describe what happened.

Include correlation identifiers to track related events.

Document events thoroughly since they are the integration contract.

12.2 Choose Appropriate Granularity

Event granularity affects system characteristics.

Fine-grained events provide flexibility but increase volume and complexity.

Coarse-grained events reduce volume but may include unnecessary data.

Consider subscriber needs when choosing granularity.

Different event types can have different granularities.

12.3 Handle Failures Explicitly

Failures are inevitable in distributed systems.

Design for at-least-once delivery with idempotent handlers.

Use dead letter queues for unprocessable events.

Implement retry with exponential backoff.

Monitor delivery and processing failures.

Plan for message loss in at-most-once systems.

12.4 Manage Schema Evolution

Events will evolve over time.

Use schema registries to manage event schemas.

Prefer backward-compatible changes.

Coordinate breaking changes across publishers and subscribers.

Support multiple versions during transitions.

12.5 Consider Ordering Requirements

Ordering affects correctness and performance.

Use partition keys when order matters for related events.

Accept weaker ordering when possible for better scalability.

Document ordering guarantees and requirements.
Design handlers to tolerate reordering when appropriate.

12.6 Monitor Event Flow

Visibility into event flow is essential.
Track publication and consumption rates.
Monitor delivery latency.
Alert on delivery failures and growing backlogs.
Log events for debugging and audit.

12.7 Secure Event Infrastructure

Security requires attention across the system.
Authenticate publishers and subscribers.
Authorize access to topics and events.
Encrypt sensitive event content.
Audit access and operations.
Validate event content before processing.

12.8 Design for Scale

Anticipate growth in event volume and subscribers.
Partition topics for parallel processing.
Use consumer groups for subscriber scaling.
Consider event retention and storage requirements.
Test with realistic loads.

13 Common Challenges

Publish–subscribe systems present several common challenges.

13.1 Event Ordering

Maintaining event order is challenging in distributed systems.
Total ordering limits scalability.
Per-partition ordering requires consistent partitioning.
Cross-partition ordering requires additional coordination.
Subscribers may need to handle out-of-order events.

Strategies include using partition keys for related events, accepting eventual consistency, implementing reordering in subscribers, and using timestamp-based reconciliation.

13.2 Exactly-Once Delivery

Exactly-once delivery is difficult in distributed systems.

Network failures cause uncertainty about delivery.

Retries may cause duplicates.

Distributed transactions add complexity and reduce availability.

Strategies include designing idempotent handlers, using deduplication based on event identifiers, implementing transactional outbox patterns, and accepting at-least-once with idempotent processing.

13.3 Subscriber Backpressure

Slow subscribers can cause problems.

Buffering has limits and costs memory.

Dropping events may be unacceptable.

Blocking publishers couples their performance.

Strategies include scaling subscriber capacity, using consumer groups for load distribution, implementing flow control protocols, and buffering with spillover to disk.

13.4 Event Schema Evolution

Changing events affects publishers and subscribers.

Breaking changes require coordinated deployment.

Backward compatibility constrains evolution.

Multiple versions complicate processing.

Strategies include additive-only changes when possible, schema registries with compatibility checking, parallel running of old and new versions, and transformation layers for compatibility.

13.5 Debugging and Tracing

Asynchronous, decoupled communication complicates debugging.

Cause and effect are separated in time and space.

Multiple subscribers make tracing complex.

Failures may be silent.

Strategies include correlation identifiers flowing through events, distributed tracing integration, comprehensive logging of publication and consumption, and event replay for debugging.

13.6 Testing

Testing publish–subscribe systems requires special attention.

Asynchronous behavior is harder to test than synchronous.

Multiple subscribers create complex scenarios.

Timing dependencies cause flaky tests.

Strategies include synchronous testing modes for unit tests, explicit waiting for event processing in integration tests, test fixtures that control timing, and contract testing between publishers and subscribers.

13.7 Operations and Monitoring

Operating publish–subscribe infrastructure requires attention.

Broker failures affect all connected components.

Backlog growth indicates processing problems.

Message loss may be silent.

Strategies include redundant broker infrastructure, comprehensive monitoring of queues and delivery, alerting on anomalies, and regular testing of failure scenarios.

14 Conclusion

The publish–subscribe style provides a powerful pattern for building loosely coupled, event-driven systems. By decoupling event producers from consumers through an intermediary distribution mechanism, the style enables flexible, scalable, and evolvable architectures.

Effective publish–subscribe architecture requires attention to event design, delivery guarantees, ordering requirements, failure handling, and operational concerns. The patterns and practices described in this document provide guidance for building robust publish–subscribe systems.

The style’s fundamental insight—that indirect communication through events enables loose coupling—has proven valuable across decades of software architecture, from GUI frameworks to enterprise integration to modern event streaming platforms. Understanding publish–subscribe architecture equips architects to design effective event-driven systems across many domains.

As systems become more distributed and real-time, the publish–subscribe style becomes increasingly relevant. Its ability to connect components without direct dependencies makes it essential for building the flexible, responsive systems that modern applications demand.

References

- Clements, P., Bachmann, F., Bass, L., Garlan, D., Ivers, J., Little, R., Merson, P., Nord, R., & Stafford, J. (2010). *Documenting Software Architectures: Views and Beyond* (2nd ed.). Addison-Wesley Professional.
- Hohpe, G., & Woolf, B. (2003). *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*. Addison-Wesley Professional.

- Eugster, P. T., Felber, P. A., Guerraoui, R., & Kermarrec, A.-M. (2003). The many faces of publish/subscribe. *ACM Computing Surveys*, 35(2), 114–131.
- Kleppmann, M. (2017). *Designing Data-Intensive Applications*. O'Reilly Media.
- Narkhede, N., Shapira, G., & Palino, T. (2017). *Kafka: The Definitive Guide*. O'Reilly Media.
- Richardson, C. (2018). *Microservices Patterns*. Manning Publications.
- Vernon, V. (2013). *Implementing Domain-Driven Design*. Addison-Wesley Professional.