Publish–subscribe pattern

From Wikipedia, the free encyclopedia

In software architecture, **publish–subscribe** is a messaging pattern where senders of messages, called publishers, do not program the messages to be sent directly to specific receivers, called subscribers, but instead characterize published messages into classes without knowledge of which subscribers, if any, there may be. Similarly, subscribers express interest in one or more classes and only receive messages that are of interest, without knowledge of which publishers, if any, there are.

Pub/sub is a sibling of the message queue paradigm, and is typically one part of a larger message-oriented middleware system. Most messaging systems support both the pub/sub and message queue models in their API, e.g. Java Message Service (JMS).

This pattern provides greater network scalability and a more dynamic network topology, with a resulting decreased flexibility to modify the Publisher and its structure of the data published.

Contents

Message filtering

In the pub/sub model, subscribers typically receive only a subset of the total messages published. The process of selecting messages for reception and processing is called *filtering*. There are two common forms of filtering: topic-based and content-based.

In a **topic-based** system, messages are published to "topics" or named logical channels. Subscribers in a topic-based system will receive all messages published to the topics to which they subscribe, and all subscribers to a topic will receive the same messages. The publisher is responsible for defining the classes of messages to which subscribers can subscribe.

In a **content-based** system, messages are only delivered to a subscriber if the attributes or content of those messages match constraints defined by the subscriber. The subscriber is responsible for classifying the messages.

Some systems support a **hybrid** of the two; publishers post messages to a topic while subscribers register content-based subscriptions to one or more topics.

Topologies

In many pub/sub systems, publishers post messages to an intermediary message broker or event bus, and subscribers register subscriptions with that broker, letting the broker perform the filtering. The broker normally performs a store and forward function to route messages from publishers to subscribers. In addition, the broker may prioritize messages in a queue before routing.

Subscribers may register for specific messages at build time, initialization time or runtime. In GUI systems, subscribers can be coded to handle user commands (e.g., click of a button), which corresponds to build time registration. Some frameworks and software products use xml configuration files to register subscribers. These configuration files are read at initialization time. The most sophisticated alternative is when subscribers can be added or removed at runtime. This latter approach is used, for example, in database triggers, mailing lists, and RSS.

The Data Distribution Service (DDS) middleware does not use a broker in the middle. Instead, each publisher and subscriber in the pub/sub system shares meta-data about each other via IP multicast. The publisher and the subscribers cache this information locally and route messages based on the discovery of each other in the shared cognizance.

History

One of the earliest publicly described pub/sub systems was the "news" subsystem of the Isis Toolkit, described at the 1987 Association for Computing Machinery (ACM) Symposium on Operating Systems Principles conference (SOSP '87), in a paper "Exploiting Virtual Synchrony in Distributed Systems. 123–138".[1]

Advantages

Loose coupling

Publishers are loosely coupled to subscribers, and need not even know of their existence. With the

topic being the focus, publishers and subscribers are allowed to remain ignorant of system topology. Each can continue to operate normally regardless of the other. In the traditional tightly coupled client–server paradigm, the client cannot post messages to the server while the server process is not running, nor can the server receive messages unless the client is running. Many pub/sub systems decouple not only the locations of the publishers and subscribers, but also decouple them temporally. A common strategy used by middleware analysts with such pub/sub systems is to take down a publisher to allow the subscriber to work through the backlog (a form of bandwidth throttling).

Scalability

Pub/sub provides the opportunity for better scalability than traditional client–server, through parallel operation, message caching, tree-based or network-based routing, etc. However, in certain types of tightly coupled, high-volume enterprise environments, as systems scale up to become data centers with thousands of servers sharing the pub/sub infrastructure, current vendor systems often lose this benefit; scalability for pub/sub products under high load in these contexts is a research challenge. Outside of the enterprise environment, on the other hand, the pub/sub paradigm has proven its scalability to volumes far beyond those of a single data centre, providing Internet-wide distributed messaging through web syndication protocols such as RSS and Atom (standard). These syndication protocols accept higher latency and lack of delivery guarantees in exchange for the ability for even a low-end web server to syndicate messages to (potentially) millions of separate subscriber nodes.

Disadvantages

The most serious problems with pub/sub systems are a side-effect of their main advantage: the decoupling of publisher from subscriber.

Inflexible Semantic coupling

The structure of the data published must be well defined, and quickly becomes rather inflexible. In order to modify the published data structure, it would be necessary to know about all the Subscribers, and either modify them also, or maintain compatibility with older versions. This makes refactoring of Publisher code much more difficult. Since requirements change over time, the inflexibility of the data structure becomes a burden on the programmer.

However, this is a common issue with any client/server architecture and is best served by versioning content payloads or topics and/or changing URL end points for backward compatibility.

In more academic terms:

[It is true that] Pub/sub systems have loose coupling within space, time and synchronization, providing a scalable infrastructure for information exchange and distributed workflows. However, pub/sub are tightly coupled, via event subscriptions and patterns, to the semantics of the underlying event schema and values. The high degree of semantic heterogeneity of events in large and open deployments such as smart cities and the sensor web makes it difficult to develop and maintain pub/sub systems. In order to address semantic coupling within pub/sub systems the use of approximate semantic matching of events is an active area of research.[2]

Message Delivery Issues

A pub/sub system must be designed carefully to be able to provide stronger system properties that a particular application might require, such as assured delivery.

- The broker in a pub/sub system may be designed to deliver messages for a specified time, but then stop attempting delivery, whether or not it has received confirmation of successful receipt of the message by all subscribers. A pub/sub system so-designed cannot guarantee delivery of messages to any applications that might require such assured delivery. Tighter coupling of the designs of such a publisher and subscriber pair must be enforced outside of the pub/sub architecture to accomplish such assured delivery (e.g. by requiring the subscriber to publish receipt messages).
- A publisher in a pub/sub system may "assume" that a subscriber is listening when it is not. A factory may utilize a pub/sub system where equipment can publish problems or failures to a subscriber that displays and logs those problems. If the logger fails (crashes), equipment problem publishers won't necessarily receive notice of the logger failure and error messages will not be displayed or recorded by any equipment on the pub/sub system. It should be noted that this is also a design challenge for alternative messaging architectures, such as a client/server system. In a client/server system, when an error logger fails, the system will receive an indication of the error logger (server) failure. But the client/server system will have to deal with that failure by having redundant logging servers online, or spawning fallback logging servers dynamically. This adds complexity to the client and server designs and the client/server architecture as a whole. However, in a pub/sub system, redundant logging subscribers that are exact duplicates of the existing logger can be added to the system to increase logging reliability without any impact to any other equipment on the system. In a pub/sub system, the feature of assured error message logging, can be added incrementally, subsequent to implementing the simpler basic functionality of equipment problem message logging.

Pub/sub scales well for small networks with a small number of publishers and subscriber nodes and low message volume. However, as the number of nodes and messages grows, the likelihood of instabilities increases, limiting the maximum scalability of a pub/sub network. Example throughput instabilities at large scales include:

- Load surges—periods when subscriber requests saturate network throughput followed by periods of low message volume (underutilized network bandwidth)
- Slowdowns—more and more applications use the system (even if they are communicating on separate pub/sub channels) the message volume flow to an individual subscriber will slow
- IP broadcast storms—a local area network may be shut down entirely by saturating it with overhead messages that choke out all normal traffic unrelated to the pub/sub traffic

For pub/sub systems that use brokers (servers), the agreement for a broker to send messages to a subscriber is in-band, and can be subject to security problems. Brokers might be fooled into sending notifications to the wrong client, amplifying denial of service requests against the client. Brokers themselves could be overloaded as they allocate resources to track created subscriptions.

Even with systems that do not rely on brokers, a subscriber might be able to receive data that it is not authorized to receive. An unauthorized publisher may be able to introduce incorrect or damaging messages into the pub/sub system. This is especially true with systems that broadcast or multicast their messages. Encryption (e.g. Transport Layer Security (SSL/TLS)) can prevent unauthorized access but cannot prevent damaging messages from being introduced by authorized publishers. Architectures other than pub/sub, such as client/server systems are also vulnerable to authorized message senders that behave maliciously.

See also

- Message brokers
- PubSubHubbub, an implementation of pub/sub
- RSS, a highly scalable web-syndication protocol
- Atom (standard), another highly scalable web-syndication protocol
- Event-driven programming
- Observer Pattern
- High-level architecture
- Data Distribution Service (DDS)
- Push technology
- Usenet
- Internet Group Management Protocol

References

1. Birman, K. and Joseph, T. "Exploiting virtual synchrony in distributed systems (http://portal.acm.org/citation.cfm?id=41457.37515&coll=portal&dl=ACM&CFID=97240647&CFTOKEN=78805594)" *in Proceedings of the eleventh ACM Symposium on Operating systems principles (SOSP '87)*, 1987. pp. 123–138.
2. Hasan, Souleiman, Sean O'Riain, and Edward Curry. 2012. "Approximate Semantic Matching of Heterogeneous Events." (http://www.edwardcurry.org/publications/Hasan_DEBS_2012.pdf) In 6th ACM International Conference on Distributed Event-Based Systems (DEBS 2012), 252–263. Berlin, Germany: ACM. "DOI" (http://dx.doi.org/10.1145/2335484.2335512).

External links

- XMPP XEP-0060: Publish-Subscribe (http://xmpp.org/extensions/xep-0060.html)
- For an open source example which is in production on MSN.com and Microsoft.com, see Distributed Publish/Subscribe Event System (http://www.codeplex.com/pubsub)
- Python PubSub (http://pubsub.sf.net) a Python Publish-Subscribe broker for messages *within* an application (NOT network)
- The OMG DDS portal (http://portals.omg.org/dds)
- libpubsub-cpp (http://github.com/cinemast/libpubsub-cpp) a topic based Publish/Subscribe framework implemented in C++
- Publish Subscribe example in C++ (http://rtmatheson.com/2010/03/working-on-the-subject-observer-pattern/)
- Synapse is a C++ framework that implements a Publish-subscribe pattern (http://open.syn3.nl/syn3/trac/default/wiki/projects/synapse)
- Programmer Question-Answer topics tagged with "publish-subscribe" (http://stackoverflow.com/questions/tagged/publish-subscribe)

Categories: Software design patterns | Distributed computing architecture | Message-oriented middleware

---