Final paper - Internetworking IK1552
Author: Jonathan Matsson (jmatss@kth.se)
Date: 30/05 2018

# The Publish-Subscribe Pattern

# Abstract

This document explains how the Publish-Subscribe pattern can be used to benefit network services in today's internetwork. The patterns of different architectures and models are explained in details. The patterns advantages and disadvantages are weighted against each other. Services that implement the pattern to solve specific problems are looked at and how their solutions solve those problems.

# 1. Introduction

The Internet protocol was built with a end-to-end focused architecture in mind [10] and the protocol used for communication over the Internet is Transmission Control Protocol/Internet Protocol (TCP/IP) [8]. TCP/IP uses source and destination addresses to send packets which means that the sender must know the destination address before sending packets. This means that the destination must have advertised its IP address beforehand to the sender. End-to-end communications gives the endpoints control over the communication which works well if the endpoints are trustworthy [9], but that is something that never should be assumed in today's Internet. According to Van Jacobson [11] the focus for the future should be on the data and not on the end nodes. The end users does not care about where the data comes from. Their only concern is to receive the data that they requested as fast and secure as possible. This can be achieved by decoupling the end nodes and letting a middleman help with the flow of data.

This architecture is used in the Publish/Subscribe (Pubsub) pattern and the middleman is usually called a broker. By decoupling the end nodes the sender need not send data directly to the receiver. Instead the receiver actively requests the data that it wants (by subscribing). This shifts the control of the data flow from sender to receiver since the receiver is the one initiating the sending of data. This should lower the amount of unwanted data received such as spam and denial of service (DoS) attacks [4].

The end nodes in the Pubsub pattern have two options (that can be derived from the name). They have the option to either publish messages or subscribe to messages. How these options works depends on how the pattern is implemented since different implementations provides different services. Since most literature writes about sending "messages", i will use that terminology instead of "data". But a message can refer to any type of data.

# 2. Architectures

The pattern can be split into two main categories for how the publisher/subscriber matching is done: Topic-based services and Content-based services [13].

## 2.1 Topic-based

In a topic-based Pubsub pattern a topic is used to differentiate between content. The end nodes then either publish or subscribe to a specific topic.

**Publishing** in a topic-based service is the equivalent of sending a message. The message is published to a specific topic. This can be done by either sending the whole message to the broker (used in Google Cloud Pub/Sub [5, para 4]) or sending something that identifies the message so that it can be requested and sent later (used in PSIRP where a identification ID together with metadata of the message is published [1, para 3.b]).

**Subscribing** is done when a node wants to receive messages. The node subscribes to a topic and will then receive all messages that are published to that topic.

## 2.2 Content-based

In a content-based Pubsub pattern the publishers does not specify a topic. Instead the content of the message decides which subscribers that will receive the message.

**Publishing** in a content-based service is done by sending a message containing a set of key/value pairs that is used to match subscriptions [14].

**Subscribing** is done by sending a filter (used in CORBA Notification Service [17]) that is used to do pattern matching against the published messages. The broker looks through the published messages and compares the content of those messages to the filter specified by a subscriber.

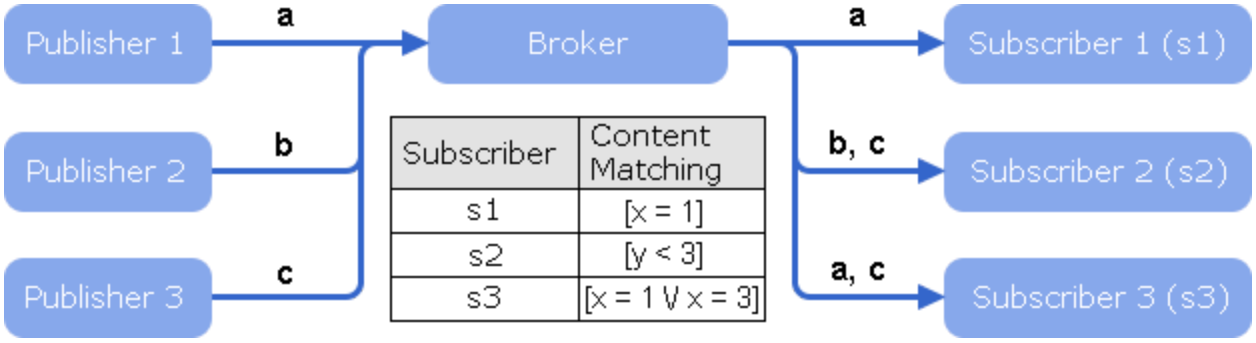


*Figure 1: The message flow in a content-based Pubsub pattern. In this example the subscribers have previously subscribed to the content they want by sending their filters to the broker. Those filters can be seen underneath the broker. The content of the messages used in this example and what subscriptions they matched can be seen in Table 1 underneath. (Adapted from Figure 1 of [12])*

*Table 1: Content of messages and what subscriptions they matched in the example used in Figure 1.*

| Sent from | Message | Message content | Sent to (matches subscription) |
|---|---|---|---|
| Publisher 1 | a | [x = 1, z = 5] | s1 (x = 1) and s3 (x = 1) |
| Publisher 2 | b | [y = 2] | s2 (y < 3) |
| Publisher 3 | c | [x = 3, y = 1] | s2 (y < 3) and s3 (x = 3) |

## 2.3 Differences between Topic-based and Content-based

Compared to a topic-based service, the content-based service gives subscribers more control over what messages they receive. By letting the subscribers filter out messages they do not want in the broking stage, the network traffic can be reduced and the receiver avoids the processing of unwanted messages [16].

A content-based service can be good when you want to filter out messages before they are sent to the subscriber. An example would be to filter out messages from a device that haven't reached a specific threshold yet. The threshold could indicate a point at which there is likely something wrong with the device and the subscribing "controller" would only be notified when that point is reached. The content-based model is also good in situations where a node wants a small amount of data from a large set of data. Letting the broker handle the filtering of the data it can greatly reduce the computing power needed by the end node.

The matching-phase is simpler in a topic-based service since the broker only has to look at, and match, one attribute between the publishers and subscribers (the topic). In contrast, a content-based model forces the broker to look through the whole message and try to match that with arbitrary large filters. Using a content-based model gives greater flexibility for the subscribers [15], while a topic-based model reduces the processing load on the broker which should lead to faster routing of messages by the broker.

# 3. Push/Pull model

The pattern can be implemented with different push or pull preferences. Publishing can be done by either pushing or pulling the message and subscribing can be done in both manners as well. This gives a total of four different combinations [25].

The use of a pull model for subscribing gives the subscriber more control over the message flow. The subscriber decides when a message should be read and it does not get interrupted by the broker when a new message is published [24]. This can be especially helpful in Internet of Things (IoT) devices where this would let the IoT devices stay in sleep mode to reduce energy consumption, but periodically waking up when they want to process messages. A push model for subscribing would let the subscribers receive new messages as soon as they are published.

Using a pull model while publishing would free the broker from storing messages. The broker would instead be able to request (pull) the message from the publisher when it is to be sent to a subscriber. In contrast, a push publisher would be able to send the message to the broker and afterwards have no responsibility to remember anything about the message. That responsibility is instead put on the broker.

# 4. Implementations

Since Pubsub is only a basic idea of how a message-routing architecture can be built, services that have implemented the Pubsub pattern differ a lot depending on how the pattern was implemented. Different services were created to solve different problems. In this section a number of services that have implemented the Pubsub pattern will be examined.

## 4.1 Blackadder

Publish-Subscribe Internet Routing Paradigm (PSIRP) was a 30 months research project where the goal was to create a *clean slate design* for the IP layer [6, para 1]. This means that the goal was to implement a new Internet Protocol to replace the current TCP/IP. According to [4], the aim with the project was to remove the unicast nature of the Internet and make multicast the standard. They also wanted to implement security and mobility directly into the foundation of the protocol. "We aspire to change the routing and forwarding fabric of the global internetwork so as to operate entirely based on the notion of information …", quote from PSIRP Vision and Use Cases [6, para 3.4].

Their solution was to replace the current IP with a protocol using a Pubsub pattern. The implementation uses Rendezvous Points (RP) in the role as brokers [3, para 2]. The RP is the equivalent of a router in today's Internet. The RPs are split into scopes. A scope is the equivalent of a network. A project called PURSUIT [19] built on the works of PSIRP and created a prototype named Blackadder.

**Publishing** in Blackadder is done by producing a rendezvous identifier (Rid) that is used to identify the message[1, para 2]. This Rid is sent to the default RP ("default router") together with metadata of the message.

**Subscribing** in Blackadder can be done by either subscribing to a scope or subscribing to a specific Rid [18]. This subscribe message is sent to the subscriber's default RP.

## 4.2 Google Cloud Pubsub

Google handles a huge amount of data. According to [5, para 1] the Google Cloud Pubsub sends over 100 million messages per second which equals 300 GB of data. Their biggest concern when building the Pubsub implementation was thus high scalability. Google Cloud Pubsub is a topic-based service with the capability of both push and pull between end nodes and the service.

The service is split into two planes: a *controlplane* and a *dataplane* [5, para 4]. The servers in the *controlplane* are responsible for distributing the end nodes to servers in the dataplane where as the servers in the *dataplane* forward messages from publishers to subscribers.

**Publishing** is done by specifying a topic and sending the message to the Google Cloud Pubsub service. The message is sent with HTTPs [20] to a *publishing forwarder* which is a server located in the dataplane. The specific publishing forwarder that the message is sent to is determined by the controlplane.

**Subscribing** is done by specifying a topic which is sent to a *subscribing forwarder* over HTTPs. The subscribing forwarder is then responsible for requesting messages from every publishing forwarder that receives messages for its specific topic.

## 4.3 Common Object Request Broker Architecture Notification Service

Common Object Request Broker Architecture (CORBA) Notification Service [26] is a service specified by the Object Management Group (OMG). CORBA can use both a topic-based and content-based architecture. The content-based implementation uses filters [17] to distribute the messages. The implementation can both push and pull messages. The service uses a *Event Channel* [26, para 2.1.2] that the end nodes connects to and communicates through.

**Publishing** when using the content-based model is done by sending key/value pairs to the Event Channel [17].

**Subscribing** is done by sending a filter containing a boolean expression that tries to match the key/value pairs of the messages [30].

See *Figure 1* and *Table 1* in Section 2.2 for an example of how a content-based Pubsub implementation works.

## 4.4 Message Queuing Telemetry Transport

Message Queuing Telemetry Transport (MQTT) is a lightweight Pubsub transport protocol [7]. According to the *Eclipse IoT Developer Survey* [21] the most common protocol used for communication with IoT devices was HTTP with MQTT as the second most common protocol. G. Montenegro, et al. [22] argue that the major reason HTTP is used more often is because it is a more well known protocol. The MQTT protocol is to be used with a client/server model where the clients publish and subscribe to messages, while the server works as the broker. The protocol realizes a topic-based service.

A MQTT message consists of two headers and a payload section. The first header is a fixed sized header that contains information about the message. It specifies what kind of message it is, for example a publish message, a subscribe message, or one of many different types of control messages [7 para 2.1]. The header also specifies the length of the message. The second header is an optional header that is used for specifying extra options for some messages types. This

header is used when specifying a topic during publishing or subscribing [7, para 3.3.2 & 3.8.2]. The payload section contains the data.

Since MQTT is a transport protocol it does not provide an implementation for the broker part of the Pubsub. Therefore the protocol is used in combination with other services that provides the broking. An example of a service that uses the MQTT protocol is Eclipse Mosquitto [23].

# 5. Problems and their solutions

The Pubsub pattern is used in implementations that aim to solve many different problems. Some of those problems are, as mentioned earlier: scalability [5, para 1], multicast [4], and data focus (instead of end node focus) [6, para 3.4]. In this section these problems will be looked at and some alternative solutions will be compared to the solutions offered by the Pubsub pattern.

## 5.1 Scalability

With the growth of the Internet and devices that communicate over networks, the scalability of services is very important. Ericsson mobility report estimates that 28 billion devices will be connected and communicate using an IP stack by the year 2021 [27]. Scalability was the biggest reason Google created the Google Cloud Pubsub [5, para 1].

The Pubsub pattern removes the unicast nature of TCP/IP [8] where the sender needs to know of all receivers. Instead a Pubsub service shifts that responsibility to a middleman which usually is more scalable and can handle more traffic than the end nodes.

Using a message broker could, if implemented incorrectly, also lead to worse performance. All the traffic goes through the broker and if the broker receives more traffic than it can handle it could cause a bottleneck.

Scalability can be helped using multicast and anycast described in Sections 5.2 and 5.3.

## 5.2 Multicast

The Pubsub pattern is design in such a way that it allows many-to-many communication. This is achieved by letting multiple nodes subscribe to the same messages and multiple nodes publish to the same topic (or send similar key/value pairs in a content-based service).

*IP Multicast* [28] is used to send one packet to multiple receivers. Internet Group Management Protocol (IGMP) is used to group hosts to a specific multicast address in IPv4 [2, page 20]. A packet is sent to that specific multicast address and the packet is then routed in such a way that the packet is copied and spread to the receiving hosts at the last possible moment. By multicasting the sender only needs to send one packet to all the receivers, while unicast communications would require the sender to send the same packet to all receivers one by one.

This could be used together with a Pubsub implementation where multicast could be used to send one message from the broker to multiple subscribers.

## 5.3 Data focus

According to Van Jacobson [11] the focus should in the future be changed from the end nodes to the data. This can be achieved by decoupling the end nodes and using a middleman to decide control the flow of the data. This is one of the main benefits of the Pubsub pattern. Using caches is also a good example of ways to focus on the data. Caches works like middlemen where they deliver the requested data without being the serving end node.

Anycast [29] is used to assign multiple nodes to one address. When a packet is sent to that address the packet is only sent to one of those nodes. This can be used to give multiple servers the same address so that the client requesting data does not need to specify a specific end node. Instead the client gets served the requested data from a available server. This is especially helpful with load distribution for providing scalability.

Anycast could be used in combination with a Pubsub service if the Pubsub broker consist of several servers that work in parallel.

# 6. Conclusion

I believe that Pubsub is a great contributor to the world of networking. Its greatest strength is the decoupling of the end nodes that opens up for scalability and many-to-many communication which are important goals for the future of networking. Its asynchronous nature makes it ideal for handling large amounts of traffic sent and received by multiple end nodes. The decoupling of the end nodes leads to higher cohesion and lower coupling [31]. This gives the publishers and subscribers more options on how to operate since they do not send messages directly between each other, they are tied together with a broker.

For example a low powered IoT device might want to communicate in a different way than a regular desktop computer. Since those two nodes are decoupled, they can communicate using their own preferences as long as the broker can interpret and translate those messages. This means that the IoT device could send its messages with a low overhead protocol such as MQTT [para 5.4], while the desktop computer could send its messages with a heavier weight protocol that allows more flexibility and functionality, but the two devices would still be able to communicate.

The pattern also simplifies sharing of data. Since an end node that wants some specific data does not need to connect directly to the source distributor of the data, it makes it simpler for the end node to find that data. The end node asks for the data by subscribing to the broker and the broker then finds the requested data. This removes the part where the end node beforehand must find the source distributor of the data.

The security aspect of the pattern can be both good and bad depending on the implementation. The security is increased by decoupling the end nodes and avoids them knowing about each other. But security could also be decreased since all the messages have to go through the broker. The end nodes can not be sure about what the broker does with the messages and where

they are sent. If the end nodes trust each other it would theoretically be more secure to send the messages using an end-to-end architecture.

The same argument could be made for the assurance that the messages are received by the receiving end node. When sending messages over TCP between the end nodes, the sender is notified if the messages made it to the receiver or not [32]. Since the sender (publisher) in a Pubsub pattern only communicates with the middleman (the broker), it can not be sure that the message made it to the receiver (subscriber). The middleman can report back the status of the message, but the sender can not be sure that the middleman is telling the truth.

A clean slate design of the Internet with a Pubsub architecture in mind, as described in Section 4.1, is an interesting idea. But my opinion is that it should not be done. The Pubsub pattern would be better than the current TCP/IP model for a lot of networks, but I feel that it should not be the building stones for the Internet. The pattern is great for many-to-many communications but is not designed for simple one-to-one communication, where the broker would introduce unnecessary delay and processing (unless a content-based filter is used to reduce the computing on the receiving node). The use of Pubsub would aggravate problems in *ad hoc* networks and other network structures that do not benefit from the pattern.

## References


[1] Nikos Fotiou, Dirk Trossen, and George C. Polyzos, 'Illustrating a publish-subscribe Internet architecture', Telecommunication Systems, vol. 51, no. 4, pp. 233–245, Dec. 2012. DOI: 10.1007/s11235-011-9432-5. Available: http://pages.cs.aueb.gr/~fotiou/papers/Future2009.pdf

[2] G. Maguire, "Module 8: Multicasting and RSVP", lecture notes IK1552 KTH Royal Institute of Technology, slide 20, 8 April 2018.

[3] Tarkoma Sasu, Ain Mark, and Visala Kari, 'The Publish/Subscribe Internet Routing Paradigm (PSIRP): Designing the Future Internet Architecture', Stand Alone, pp. 102–111, 2009. DOI: 10.3233/978-1-60750-007-0-102. Available: http://www.psirp.org/files/Deliverables/OverlayMulticastAssistedMobility.PDF

[4] S. Tarkoma, M. Ain and K. Visala, "The Publish/Subscribe Internet Routing Paradigm(PSIRP): Designing the Future Internet Architecture", page 120, May 2009. Available: http://www.cse.tkk.fi/fi/opinnot/T-110.6120/2012_Autumn_Future_Internet/lisatty-files/psirp.pdf

[5] Google, "Cloud Pub/Sub: A Google-Scale Messaging Service", 11 May 2018. Available: https://cloud.google.com/pubsub/architecture

[6] D. Trossen, J.Tuononen, G. Xylomenos, M. Sarela, A. Zahemzky, P. Nikander and T. Rinta-aho, "PSIRP Vision and Use Cases", 23 May 2008. Available: http://www.psirp.org/files/Deliverables/PSIRP-TR08-0001_Vision.pdf

[7] A. Banks and R. Gupta, "MQTT Version 3.1.1 OASIS Standard", 29 October 2014. Available: http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/os/mqtt-v3.1.1-os.pdf

[8] J. Postel, 'Internet Protocol', Internet Request for Comments, vol. RFC 791 (INTERNET STANDARD), Sep. 1981. Available: https://www.rfc-editor.org/rfc/rfc791.txt

[9] M. Blumenthal and D. Clark, "Rethinking the Design of the Internet: The End-to-End Arguments vs. the Brave New World", *ACM Transactions on Internet Technology*, Vol. 1, No. 1, paragraph. 1.1, August 2001. Avaliable: http://nms.lcs.mit.edu/6829-papers/bravenewworld.pdf

[10] B. Carpenter, "Architectural Principles of the Internet", Internet Request for Comments, vol. RFC 1958, paragraph 2.3, June 1996. Available: https://www.rfc-editor.org/rfc/rfc1958.txt

[11] V. Jabson, "If a Clean Slate is the solution what was the problem?", *Stanford 'Clean Slate' Seminar*, slide 2, 27 February 2006. Avaliable: http://yuba.stanford.edu/cleanslate/seminars/jacobson.pdf

[12] R. Paul, "Topic-based publish/subscribe design pattern implementation in c#-Part II (Using WCF)", 23 Mars 2009. Available: https://www.codeproject.com/Articles/34333/Topic-based-publish-subscribe-design-pattern-imp

[13] F. Rahimian, "Distributed Publish/Subscribe Systems: a Quick Survey", lecture notes ID2210 KTH Royal Institute of Technology, lecture 5, slide 5, 12 April 2012. Avaliable: https://www.kth.se/social/upload/5167e5e8f276546cb4f488df/5-Publish-Subscribe.pdf

[14] A. Carzaniga, M. J. Rutherford, and A. L. Wolf, 'A routing scheme for content-based networking', in IEEE INFOCOM 2004, 2004, vol. 2, pp. 918–928 vol.2. DOI: 10.1109/INFCOM.2004.135697. Available: http://www.inf.usi.ch/carzaniga/papers/crw_infocom04.pdf

[15] F. Fabrte, F. Llirbat, J. Pereira and D. Shasha, "Efficient Matching for Content-based Publish/Subscribe Systems", Technical report, INRIA, September 2000. Available: https://pdfs.semanticscholar.org/c597/59cb4fa2a5fa448aa404f2f409f1544accae.pdf

[16] C. McHale, "Publish and Subscribe Services", paragraph 22.3.4.1. Available: http://www.ciaranmchale.com/corba-explained-simply/publish-and-subscribe-services.html#sect:notify-service:remove-filters

[17] Oracle, "CORBA Notification Service API Reference", paragraph "*Structured Event Fields, Types, and Filters*". Available: https://docs.oracle.com/cd/E13203_01/tuxedo/tux80/notify/evnt_api.htm#1023894

[18] FP7-PURSUIT, “Blackadder Application Programming Interface”, page 3-4, 7 November 2012. Available: https://github.com/fp7-pursuit/blackadder/blob/master/api.pdf
[19] FP7-PURSUIT. Available: http://www.fp7-pursuit.eu/PursuitWeb/
[20] Google, “Frequently Asked Questions”, paragraph ‘*How are messages sent to push endpoints secured?*’, 16 May 2018. Available: https://cloud.google.com/pubsub/docs/faq#security
[21] Eclipse IoT Working Group, IEEE IoT and AGILE IoT, “Eclipse IoT Developer Survey”, Page 24, April 2016. Available: https://iot.ieee.org/images/files/pdf/iot-developer-survey-2016-report-final.pdf
[22] G. Montenegro, S. Cespedes, S. Loreto and R.Simpson, “H2oT: HTTP/2 for the Internet of Things”, paragraph 2.1, 8 July 2016. Available: https://tools.ietf.org/html/draft-montenegro-httpbis-h2ot-00#section-2.1
[23] R. A. Light, "Mosquitto: server and client implementation of the MQTT protocol," *The Journal of Open Source Software*, vol. 2, no. 13, May 2017, DOI: 10.21105/joss.00265
[24] Google, “Pull Subscriber Guide”, paragraph “*Asynchronous Pull”*. Available: https://cloud.google.com/pubsub/docs/pull#asynchronous-pull
[25] Erlang.org (Ericssons Telecom AB), “Event Service”, paragraph 5.2. Available: http://www1.erlang.org/documentation/doc-4.7.3/lib/orber-2.0/doc/html/ch_event_service.html
[26] Object Managment Group, “Notification Service Specification”, October 2014. Available: https://www.omg.org/spec/NOT/About-NOT/
[27] Ericsson, “Ericsson Mobility Report”, page 10, November 2015. Available: https://www.ericsson.com/assets/local/news/2016/03/ericsson-mobility-report-nov-2015.pdf
[28] S. E. Deering, “Host extensions for IP multicasting”, Internet Request for Comments, vol. RFC 1112 (INTERNET STANDARD), Aug. 1989. Available: https://www.rfc-editor.org/rfc/rfc1112.txt
[29] J. Abley and K. Lindqvist, “Operation of Anycast Services”, Internet Request for Comments, vol. RFC 4786, December 2006. Available: https://www.rfc-editor.org/rfc/rfc4786.txt
[30] Oracle, “CORBA Notification Service API Reference”, paragraph “*Parameters Used When Creating Subscriptions*”. Available: https://docs.oracle.com/cd/E13203_01/tuxedo/tux80/notify/evnt_api.htm#1110161
[31] L. Lindbäck, “A First Course in Object Oriented Development, A Hands-On Approach”, paragraph 5.2, April 11 2017.
[32] G. Maguire, “Module 5: TCP, HTTP, RPC, NFS, X”, lecture notes IK1552, KTH Royal Institute of Technology, slide 5, 25 Mars 2018.