Survey of Publish Subscribe Event Systems

Ying Liu Beth Plale

Computer Science Dept.
Indiana University
Bloomington, IN 47405-7104
{yingliu,plale} @cs.indiana.edu

TR 574

Abstract

The Internet has changed considerably the scale of distributed systems, motivating the demand for more flexible communication models and systems. Individual point-to-point and synchronous communications, which tend to lead to rigid and static applications, are making way for the more loosely coupled interaction such as is supported by a publish-subscribe paradigm.

In this paper, we define a taxonomy for comparing and contrasting publish subscribe systems, citing examples from the systems included in the survey. We then survey existing publish subscribe systems, and discuss their features with respect to the taxonomy. The appendix contains a code example that demonstrates use of a typical publish subscribe system.

1 Introduction

The Internet has considerably changed the scale of distributed systems. Distributed systems now involve thousands of entities – potentially distributed worldwide – whose location and behaviors may vary throughout the lifetime of the system. These constraints motivate the demand for more flexible communication models and systems, which reflects the dynamic and decoupled nature of the applications. Individual point-to-point and synchronous communications tend to lead to rigid and static applications, and make the development of dynamic large scale applications cumbersome. To reduce the burden of application designers, the glue between the different entities in such large scale settings should rather be provided by a flexible loosely coupled communication infrastructure.

The publish-subscribe paradigm is receiving increased attention for the loosely coupled form of interaction it provides in large scale settings. In general, subscribers register their interests in a topic or a pattern of events and then asynchronously receive events matching their interest, regardless of the events' publisher. The strength of an

event-based interaction style is drawn from full decoupling in time, space and flow between publishers and subscribers. However, because of the multiplicity of these systems and prototypes, it is rather difficult to capture their commonalities, and to draw a sharp distinction between their main variations [11].

In this paper, we first define a taxonomy for comparing and contrasting publish subscribe systems, citing examples from the systems included in the survey. Then we give several examples of existing publish subscribe systems. The systems examined in this survey are drawn from standalone publish-subscribe systems in the public domain. Thus though systems like Legion [14], MPI, and Gnutella [13] may provide support for data streams, their purpose is broader than standalone support for the communication paradigm so they are not included here. At last, we present a simple code example of a publisher and subscriber using the Siena [5] system as the example API.

2 Mechanisms of Publish-Subscribe Systems

Publish-subscribe sytems differ on a number of fundamental characteristics. The most popular decomposition of systems is into the general categories of subject-based or content-based systems. Systems differ on their architectures as well. Publish-subscribe systems can be push-based, pull-based, or both. In push-based, messages are automatically broadcast to subscribers. This model provides tight consistency and stores minimal data. Pull-based models can be more responsive to user needs. Publish-subscribe systems must address scalability, in terms of subscription management, and in terms of efficient *matching* of an event. against a large numbe rof subscribers, and efficient *distribution* of events.

In this section we introduce a taxonomy of key features of publish-subscribe system consisting of:

- Subject-based versus content-based systems,
- System architectures,
- Matching algorithms,
- Multicast algorithms, and
- Reliability and security.

2.1 Subject-based vs. Content-based

There are two general categories of publish-subscribe systems, subject-based or content-based. In *subject-based systems*, a message belongs to one of a fixed set of what are variously referred to as groups, channels, or topics. Subscription targets a group, channel, or topic, and the user receives all events that are associated with that group. Brokering a connection between publishers and subscribers is the act of connecting a channel supplier with a channel consumer, similar to the reader-writer problem in that the buffer is the communication medium. For example, in a subject- based system for stock trading, a participant could select one or two stocks then subscribe based on stock name if that were one of valid groups. If it were only PE-ratio she were interested in, she would likely receive much more information than needed.

Content-based systems, on the other hand, are not constrained to the notion that a message must belong to a particular group. Instead, the decision of to whom a message is directed is made on a message-by-message basis based on a query or predicate issued by a subscriber. The advantage of a content-based system is its flexibility. It provides the subscriber just the information he/she needs. The subscriber need not have to learn a set of topic names and their content before subscribing. Returning to our stock trading example, it is not necessary for a participant to spend time learning information of all stocks. Instead she can simply list the conditions for her ideal stock then subscribe using those conditions, for instance, $\{$price(15,20), PE-ratio< 20, Earning-Per-Share> 0.5 $\}$. The system will automatically select relevant stocks for her, for example, Oracle and IBM. If she were to need *Average Volume* information of the stocks she holds, she could subscribe using a predicate such as $\{$attribute=avgVol$\}$. A disadvantage of content-based systems is the burden it places on the underlying system to match messages to the subscriptions. The number of unique subscriptions can be orders of magnitude larger than the number of groups that must be managed in the subject-based system. Hence matching must be done extremely efficiently. The problem of matching arriving events against a large number of queries has also been addressed in the database community with data dissemination systems such as Xyleme [19] and XFilter [2].

2.2 System Architecture

An event system is a distributed communication paradigm consisting of software components that realize the event service and a unique communication paradigm. The software components can be event servers, event clients, or both. The architecture of publish-subscribe system can be classified into the general categories of *client-server* or *peer-to-peer*.

2.2.1 Client-Server Model

In the *client-server model,* a component serves as an event server or an event client. Event servers receive events, possibly store them, and forward them. Event servers communicate with other event servers to achieve such properties as scalability. Clients act as publishers, subscribers, or both. The general kinds of topologies [15] by which servers are related include:

- Star topology (Centralized server)
- Hierarchical topology
- Ring topology
- Irregular polygon topology

Star (Centralized server) topology. An event system with a centralized server topology relies upon a single event server to broker between publishers and subscribers. As is shown in Figure 1, four providers, P1 thru P4, publish events to be received by subscribers S1 thru S4. A subscriber may receive events from any of the providers; the key notion is the existence of a single server that brokers between publishers and subscribers. This simple topology does not scale well and such is not realized in practice.

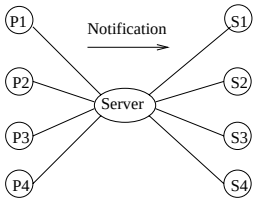

Figure 1: Centralized server topology

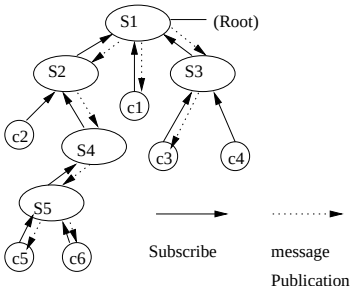

Figure 2: Hierarchical server topology

Hierarchical server topology. The hierarchical topology is distinguished by a hierarchical relationship between event servers. As shown in Figure 2, each server, identified as S1 thru S5, serves a number of clients, C1 thru C6. Clients can either be publishers or subscribers. Event servers connect to a parent server. In this topology, the communication between server-server and client-server follows the same protocol. Thus a server does not distinguish between other servers and its clients. The purpose of the hierarchical organization is scalability. A parent server will receive published events and subscriptions from all of its clients but will forward to its subtree only events destined for that subtree. The parent server acts as a gatekeeper in this regard, keeping general traffic off the subtree.

Ring topology In the ring topology, as shown in Figure 3, the servers exist in a peer-to-peer relationship with one another, and the server connection graph is a ring. The communication between servers is via a bidirectional communication protocol for exchanging subscriptions and notifications. The server-server protocol is different from the client-server protocol in the type and amount of information exchanged. For a server-server communication, the two end nodes maintain information for each other. But for a client-server connection, a client can generate subscription and be the final recipient for the published message, but the server on the other hands serves as an access point or router to pass on messages.

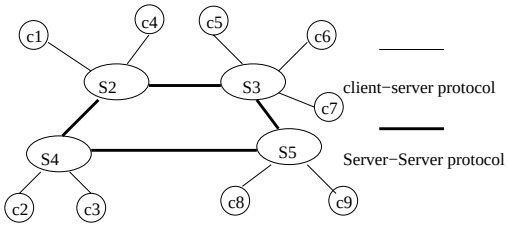

Figure 3: Ring topology

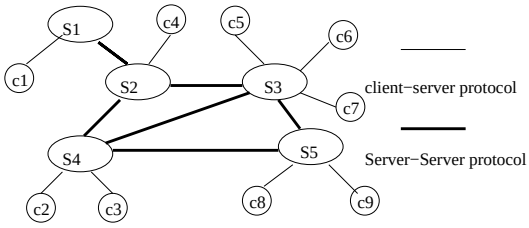

Figure 4: Irregular polygon topology

Irregular polygon topology The irregular polygon topology is a generalized ring topology absent of the constraint that all servers connect as a ring. Thus the network graph is a generic graph (see Figure 4). Similar to the ring topology, this topology allows bi-directional communications between two servers.

2.2.2 Peer-to-Peer Model

In the *peer-to-peer model*, all nodes are equal, as shown in Figure 5. That is, each node can act as a publisher, subscriber, root of a multicast tree, internal node of a multicast tree, or any reasonable combination thereof. That is there are no server nodes or client nodes in this model. Some of the server functionality(*i.e.* persistence, transactions, security) is embedded as a local part of each node.

2.3 Matching Algorithm

The matching algorithm controls the way in which events are delivered to subscribers. In subject-based systems, where messages are grouped into logical units in the form of a group, channel, or topic, the purpose of the matching algorithm is to manage these channels. Managing channels can be accomplished in a straightforward manner through a centralized server that does a lookup on the topic ID of an arriving message to determine the connection information for the subscribers. More complex matching

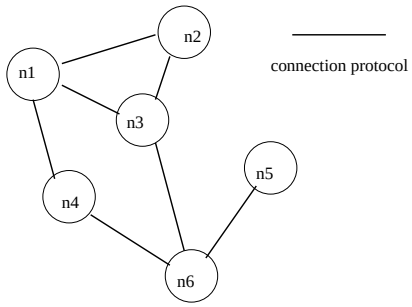

Figure 5: Peer-to-peer model

algorithms also exist. Bayeux [26] uses a hashed-suffix mesh algorithm [20] to locate subscribers and routes messages across arbitrarily large networks while using a routing map with size logarithmic to the network name-space at each hop [26]. ECho [7] establishes a direct connection between a publisher and subscriber at the time a subscriber subscribes to a channel. In this way, the centralized server is only employed at connection establishment; published events are efficiently routed directly from publisher to the subscribing clients.

Content-based systems, on the other hand, have no notion of a group, as the unit to which one subscribes as in subject-based systems. Hence these systems require different approaches of matching messages to subscribers. The consequence of the additional flexibility is that the number of unique subscriptions may be orders of magnitude larger than the number of groups that must be managed in the subject-based system. Many content-based systems use a matching tree algorithm. This algorithm preprocesses a set of subscriptions into a matching tree, with each node a partial condition on the attributes of a predicate. Each lower level of the tree is a refinement of the test performed at the next higher level. Subscriptions are resident at the leaves of the tree. Upon arrival of an event, the subscriptions matching the event are found by navigating the decision tree starting at the root. This algorithm's sub-linear time complexity with respect to the number of subscriptions, and linear space complexity is efficient for most publish-subscribe systems [1]. Other matching algorithms are considered in the discussions of the individual systems.

2.4 Event distribution scheme

Publish-subscribe systems must support a large number of geographically distributed publishers and subscribers. As the system scales, and the number of subscribers to a channel or topic grows, the need arises for some forms of efficient event distribution. This is particularly true for subject-based systems where the number of topics is small relative to the number of subscribers. The solution is often a form of software-based multicast. Multicast in a publish-subscribe system is the broadcasting of a message from one broker to subscribers that are associated with one of a group of brokers, usually in such a way that membership is transparent to the sender.

Publish-subscribe systems must support a large number of geographically distributed

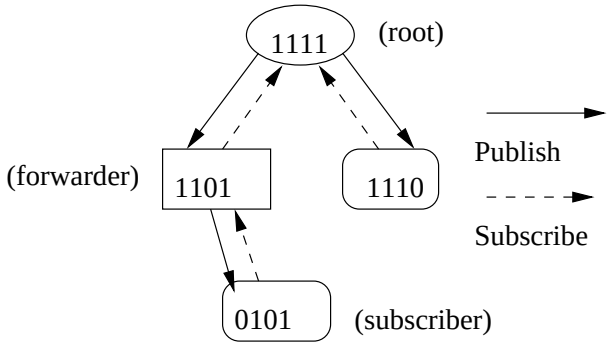

Figure 6: Simple example for Multi-cast tree

publishers and subscribers, so efficient communication between these brokers is paramount. Considering that brokers are often connected via relatively slow WAN (compared to fast LAN) and a publish-subscribe system needs good scalability, multicast algorithm's known good efficiency is crucial for the whole system.

For a subject-based system, a multicast algorithm can define a multicast group and corresponding multicast tree per subject. When a subscriber subscribes to this subject, the subscriber node is added to the corresponding multi-cast tree. When an event for a subject is published, it is disseminated using the multi-cast tree. This is illustrated in Figure 6. In this part of multi-cast tree, each node has a binary code as its identification. When a node publishes or forwards message, it sends a message only to the nodes whose binary code is one-bit different from its own code. So in this graph, node (1111) can send a message to nodes (1101 and 1110). However, node (1111) can not directly send a message to node (0101), although node (0101) is the real subscriber for this message.

But this subject-based multicast technique cannot be readily applied to a content-based system because in a content-based system, each subscriber may have a unique subscription, and therefore, we can not easily find common characters to group many subscribers together.[1] We discuss some novel and efficient distributed multicast algorithms for content-based systems in the context of the surveyed systems.

2.5 Reliability

Publish-subscribe systems provide various guarantees regarding reliability and fault tolerance ranging from *best-effort* to *guaranteed and timely*. Some systems are based on TCP and provide the guarantees that TCP provides, namely reliable point-to-point

[1] To naively map these subscribers into groups may require a number of groups exponential in the number of subscribers (i.e 2^N).

byte stream protocol. In the Scribe system, Tapestry, which is built on top of UDP, can repair the multicast tree through periodically sending heartbeat messages. Thus, Scribe guarantees fault tolerant delivery in the presence of failed brokers. Other systems use a mesh network [26]. The redundancy of a mesh network can guarantee the reliable delivery when a path is broken due to a faulty broker. Bayeux and Gryphon achieve reliability in this manner.

2.6 Security

A publish-subscribe system must handle information dissemination across distinct authoritative domains, heterogeneous platforms and a large, dynamic population of publishers and subscribers. Such environments raise new security concerns[24]. Our discussion in this section focuses on content-based systems because this kind of system is more vulnerable to security problems. The key aspects of security can be defined by example, where the scenario is Alice who sends a message via a publish-subscribe system to Bob:

- *Authentication:* Bob wants to make sure Alice sent the message
- *Confidentiality:* Alice wants to ensure only Bob can read the message
- *Integrity:* Alice wants to ensure Bob receives the message, and exactly the message she sent
- *Accountability:* Bob wants to prove only Alice could have sent the message

Authentication establishes the identity of the originator of an action. In the case of our example, Bob wants to ensure Alice is the sender of the message. But content-based systems allow anonymous subscribers, implying the content should be authenticated instead of the sender. Public Key Infrastructure (PKI) can be used for this, however, the overhead of authenticating every message is high.

Confidentiality is the ability to keep others from accessing messages. In traditional situation, Alice wants only Bob to read the message. But in publish-subscribe systems, there are different parties involved: a subscriber, a publisher, and servers, the latter of whom are needed to route the message based on its content. So complete confidentiality is difficult to obtain. A promising technique for this problem may be *computing with encrypted data*[18].

Integrity is the requirement of keeping the message in its original form. For instance, ensuring that the message Bob received is identical to the one Alice sent. Normally, cryptography is used to maintain the integrity of information. For publish subscribe systems there are different integrity problems: Information integrity, subscription integrity and service integrity.

Accountability can easily be satisfied if explicit addresses are known. For example, if Bob and Alice know each other, Alice can easily prove only Bob could have got the message. However, no direct relationship between publisher and subscriber exists in publish-subscribe systems so it is impossible for a publisher to know which subscribers can receive the message. One solution for this accountability problem is that publishers can sell keys to subscribers and with those keys, subscribers can decrypt selected data. An alternative is to ensure credibility of the publish-subscribe infrastructure.

3 Surveyed Publish-Subscribe Systems

In this section, we survey eight existing publish-subscribe systems and discuss them in the context of the taxonomy developed earlier. Their characteristics are summarized in Table 1).

3.1 Gryphon

Gryphon [3, 17] is targeted toward the distribution of large volumes of data in real-time to thousands of clients distributed throughout a large public network. It is a content-based publish subscribe system with an architecture that we categorize as client-server model. Events are matched to subscribers using a matching tree that is constructed in the preprocessing phase. The matching tree consists of nodes that are expressions evaluated over attributes, and edges that are traversed as a result of the evaluation of the expression. Each lower-level of the tree is a refinement of the condition evaluated at the higher level; the leaves of the tree are individual subscriptions. Matching occurs by traversing the tree starting at the root; at each node, the expression is evaluated and the edges are followed that are consistent with the result (there is often more than one edge.) The leaves that are ultimately visited correspond to the subscriptions that match the event.

Multicast is performed by the *link matching algorithm*. In this algorithm, brokers are assembled in a decision tree which an individual broker uses to determine which subset of its neighbors it should send an event, that is, it determines the subset of links along which it should transmit the event [17]. Gryphon uses a broker organization protocol to organize servers into cells for fault tolerant delivery. Servers belonging to a cell are fully connected. Redundant links are constructed between cells.

Gryphon provides a full suite of security features for client authentication including: simple (telnet-like) password, mutual password authentication, asymmetric password-certificate SSL authentication and symmetric certificate SSL. Clients negotiate an authentication protocol with a server upon connection. Access Control Lists(ACLs) are used to limit the topics to which an authenticated client may publish messages or subscribe. Both positive and negative access controls may be specified for any region of the topic hierarchy.

3.2 Scribe

Scribe [6, 22, 21] is a scalable subject-based system. Its fully decentralized architecture is characterized as generic peer to peer model. It is built on top of Pastry [21], a generic peer-to-peer object location and routing substrate overlayed on the Internet. As such, Scribe can leverage Pastry for features such as reliability, fault tolerance and multicast. Scribe and Pastry are fully decentralized in that all decisions are based on local information and that each node has identical capabilities. Each node can act as a publisher, a root of a multicast tree, an subscriber to a topic, a node within a tree, or any sensible combination thereof [22]. Self organization is the quality where each node is able to route client requests and interacts with local instances of one or more applications.

Scribe's matching algorithm is based on numeric keys and nodeIDs. The numeric key is a number which can be used to identify the topic. Each node in the network has a unique numeric identifier(nodeId). When presented with a message and a numeric key, a Pastry node efficiently routes the message to the node with a nodeId that is numerically closet to the key, among all currently live Pastry nodes. The expected number of routing steps is O($\log N$) where N is the number of Pastry nodes in the network. At each Pastry node along the route that a message takes, the application is notified and may perform application-specific computations related to the message [21].

Scribe uses group-based multicast algorithm. It creates a multicast tree, rooted at a rendezvous point, to disseminate the events published in the topic. The rendezvous point is the node whose nodeId is numerically closest to the numeric key. The multicast tree is created using a scheme similar to reverse path forwarding. The tree is formed by joining the Pastry routes from each subscriber to the rendezvous point. Subscriptions to a topic are managed in a decentralized manner to support large and dynamic sets of subscribers [6].

In Pastry, each non-leaf node in the tree sends a periodical heartbeat message to its children. Through these heartbeat messages, a child can detect whether its parent is live. If it suspects its parent has failed, it will call Pastry to route a Subscribe message with the topicId. Then it will get the new parent, thus repairing the multi-cast tree. Based on Pastry, Scribe can provide fault-tolerance delivery. However, Scribe just guarantees best-effort delivery. For reliable and ordered delivery events, we need stronger reliability models on top of Scribe.

Three versions exist for Pastry: FreePastry, SimPastry and Vispastry. FreePastry is Java based. SimPastry is an implementation of Pastry provided by Mircrosoft Research and VisPastry means the package of Pastry/Scribe Visualizer which is often included in the package of SimPastry. But we can get separate version for these two packages now. SimPastry and VisPastry are C based.

3.3 Bayeux

Bayeux [26] is an efficient application-level multicast system that scales to arbitrarily large receiver groups while tolerating failures in routers and network links. It is based on Tapestry [25] which has the decentralized peer-to-peer architecture. Each Tapestry node can assume the roles of server (which stores and serves objects), router (which forwards messages), and client (which serves as originator of requests). Its matching algorithm is very similar to the hashed-suffix mesh mechanisms[20].

It is novel in that it allows messages to locate objects and route to them across an arbitrarily-sized network, while using a routing map with size logarithmic to the network name-space at each hop.

Bayeux provides, on top of Tapestry, an application level multicasting protocol to organize receivers into a multicast tree rooted at the source. The Tapestry unicast routing underneath provides a natural base for application-level multicasting by forwarding packets according to suffixes of listener node IDs. A multicast system needs only to duplicate a packet when the receiver node identifiers become divergent in the next digit. In addition, the maximum number of hops taken in the overlay network by

such a delivery mechanism is bounded by the total number of digits in Tapestry node IDs.

In Tapestry, each entry in the neighbor map keeps secondary neighbors in addition to the closest primary neighbor. Leveraging this redundancy, Bayeux can provide reliable delivery when the primary route broker fails. Tapestry Release 1.0 is now available, which is java based.

3.4 Siena

Siena [5] is a content-based scalable event-notification service. Its architecture is categorized as a client-server model in that two types of clients, publishers and subscribers, exchange messages through a Siena server. Publishers connect to a Siena server to publish events they want to make the world aware of, and subscribers connect to the server to establish subscriptions, in which they specify the set of messages they are interested in receiving. Matching is accomplished at the server with a Binary Decision Diagram [4], a variation of the matching tree algorithm discussed in Section 2.3.

To use the system, a subscriber must implement a Notifiable interface. Siena delivers notifications to a subscriber by invoking the notify method on the subscriber object. 'Notifiable' has two variants for the notify method: notify(Notification n) is called to notify a single notification, while notify(Notification s[]) is called to notify a sequence of notifications. Siena's routing paths for notifications are set at time of subscription. A new subscription is stored and forwarded from the originating server to all servers in the network. This forms a tree that connects subscriber with servers. Notifications are then routed towards the subscriber following the reverse path of the tree. Siena provides a Java-based version and a C++-based version.

We have an small code example of a publish-subscribe system using Siena system. We have Interest Generator class as publisher and InterestReceiver class as subscriber. In that example, subscriber subscribes with the filter of {make='Ford', year>1999}, publisher keeps publishing events {make='Ford', model='Focus', year=2000, color='read'}, {make='Ford', model='Tauras',year=2001, color='silver'}, and {make='Buick', model='Century', year=2000, color='white'}. Subscriber gets events he needs and prints them out. I used several publishers to sent events to subscriber at the same time to see how fast the subscriber(receiver) can receive messages.

3.5 NaradaBrokering

NaradaBrokering [12] is a distributed event brokering system for wide area applications where many distributed cooperating broker nodes are required. NaradaBrokering can be categorized as a content-based publish subscribe system with a hierarchical topology of servers for event dissimenation within a cell and a peer to peer graph between cells. Matching is accomplished through the construction of a matching tree from the content of subscriptions. When an event arrives, the matching tree is traversed to locate matched subscribers. NaradaBrokering provides additional matching engines for: SQL 92 queries based on the JMS specification and XML attribute-value pairs for topic subscription. Dissemination is through software multicast.

Routing is accomplished through shortest path computations. Every broker, either targeted or en route to one, computes the shortest path to reach target destinations, considering only those links and brokers that have not failed or been failure-suspected.[12] The routing for NaradaBrokering system is near optimal since for every event the targeted set of brokers are usually the only ones involved in dissemination.

A goal of NaradaBrokering is to provide a unified messaging environment that integrates Grid Services, JMS and JXTA. Additionally, it can serve as a gateway between centralized systems like JMS compliant implementations and P2P implementations such as JXTA. Release 0.85 of NaradaBrokering is available. The security framework of NaradaBrokering is under development. The wire representation of messages is serialized Java objects. NaradaBrokering supports multiple underlying transport protocols including TCP, UDP, RTP, SSL, and HTTP.

3.6 XMessages

XMessages [23] is a hybrid subject-based and content-based publish-subscribe system. First it uses lookup table to get the reference of XMessage channel for building the connection. After that, it can use SQL-like query to filter messages from this channel. This filtering is based on the content of messages. So XMessage has characteristics of both systems. Its architecture can be characterized as a client-server graph. Its use of XML for its data representation format gives XMessages flexibility in platform and language independence.

XMessages uses a *filtering matching algorithm* where simpler predicates can be applied to select attributes of a message while more complex predicates may result in the execution of an SQL-like query. This matching is similar to JMS matching, the latter of which uses message selectors based on a subset of SQL92 conditional expression syntax.

The event listener (subscriber) and publisher can interact with each other directly through an event channel as is the traditional form of communication in publish-subscribe system or can communicate with each other via third parties, or agents. For instance, a publisher can access a listener's URL to obtain its Remote reference. Then write the event, or message, to the socket referenced by the remote reference.

XMessages is a reliable event service in the sense that when the network is down and target nodes are unreachable, the service will automatically keep trying until the network connection is up and running. Reliability is further enhanced by the maintenance of a message log of messages not yet delivered so that messages are not lost if they cannot be delivered immediately. Java-based and C++-based APIs are available for XMessages [23]. XMessages uses SOAP and XML as the wire format. A JMS Bridge is also available that allows for messages to be exchanged between JMS peers and the XMessage framework.

3.7 Echo

Echo [8] is also a hybrid of subject-based and content-based event communication system. Its architecture we categorize as peer-to-peer graph topology. ECho provides an efficient lightweight implementation of CORBA-style event channels. Event channels

are created by a provider and subscribed by subscribers. The process who creates a channel is considered the contact point for that channel. Once a channel is created, it is assigned a unique channelId which includes host name and IP port number of the creating process. Other processes can use channelId to open this channel. Derived event channels extend ECho with content-based functionality. A Derived event channel for some subscriber α is an event channel whose events pass through a user-defined filter function θ on their way from publisher to subscriber α. ECho will take care of moving the filter function to all the publishers for that channel. The filter is executed locally at the source. The filter is a side-effect free function that the user writes from a limited C-style language. the binary code for the function is generated dynamically at the source. Because ECho is essentially a content-based system, matching is the straightforward mapping from topics to the IDs of channels providing the topics. In the case of typed event channels, where a channel is restricted to carrying events of the same format, matching is accomplished at time of subscription. By inquiring to ECho on a particular data type, a subscriber can receive a list of channels supporting that particular data type. This is easier than keeping track of each channel individually.

ECho, which has its roots in DataExchange[10], has efficient event transfer through binary encoding of data using Portable Binary IO (PBIO)[9].

3.8 JMS

The Java Message Service (JMS) is a vendor-agnostic Java API that allows applciations to create, send, receive and understand messages. It defines a common set of interfaces that can be used by different Message-Oriented Middleware (MOM) vendors. In this sense JMS is analogous to JDBC in that application developers use the same API to access many different systems. JMS provides two types of messaging models, publish-subscribe and point-to-point queuing. In publish-subscribe model, JMS uses topics as intermediaries and as such is a subject-based system. One producer can send a message to many consumers through a virtual channel called a *topic*. Publish-subscribe is a push-based model. JMS can be implemented as *Client-Server model*, *Peer-to-Peer model* or both. Durable subscription is an option for JMS publish-subscribe messaging model, which allow consumers to disconnect and later reconnect and collect messages that were published while they were disconnected. There are several implementations of JMS including IBM MQSeries, Progress SonicMQ, Fiorano FioranoMQ and Sun JMQ.

4 Conclusion

We have developed a taxonomy for comparing and classifying different publish-subscribe systems. This taxonomy includes subject-based versus content-based classification, system architecture, matching algorithm, multicasting algorithm, reliability, and security. Based on such a taxonomy, we surveyed several existing publish-subscribe systems such as IBM Gryphon system, Microsoft Scribe system, Bayeux, Siena, NaradaBrokering, XMessages, Echo system, and JMS. We applied our taxonomy to each system, classified them as topic-based system or content-based system, described their archi-

tectures, matching algorithms as well as multicasting algorithms, and also talked about other features for those sytems.

References

[1] Marcos Kawazoe Aguilera, Robert E. Strom, Daniel C. Sturman, Mark Astley, and Tushar Deepak Chandra. Matching events in a content-based subscription system. In *Symposium on Principles of Distributed Computing*, pages 53–61, 1999.

[2] Mehmet Altinel and Michael J. Franklin. Efficient filtering of XML documents for selective dissemination of information. In *The VLDB Journal*, pages 53–64, 2000.

[3] Guruduth Banavar, Tushar Chandra, Bodhi Mukherjee, Jay Nagarajarao, Robert E. Storm, and Daniel C. Sturman. An efficient multicast protocol for content-based publish-subscribe systems. In *International Conference on Distributed Computing Systems*, 1999.

[4] Alexis Campailla, Sagar Chaki, Edmund M. Clarke, Somesh Jha, and Helmut Veith. Efficient filtering in publish-subscribe systems using binary decision. In *International Conference on Software Engineering*, pages 443–452, 2001.

[5] Antonio Carzaniga, David S. Rosenblum, and Alexander L Wolf. Design and evaluation of a wide-area event notification service. *ACM Transactions on Computer Systems*, 19(3):332–383, August 2001.

[6] Miguel Castro, Peter Druschel, Anne-Marie Kermarrec, and Antony Rowstron. Scribe: A large-scale and decentralized publish-subscribe infrastructure - preliminary draft submitted for publication.

[7] Greg Eisenhauer. The ECho event delivery system. Technical Report GIT-CC-99-08, College of Computing, Georgia Institute of Technology, 1999. http://www.cc.gatech.edu/tech_reports.

[8] Greg Eisenhauer, Fabian Bustamente, and Karsten Schwan. Event services for high performance computing. In *Proc. 9th IEEE Intl. High Performance Distributed Computing (HPDC)*, Los Alamitos, CA, August 2000. IEEE Computer Society.

[9] Greg Eisenhauer and Lynn K. Daley. Fast heterogeneous binary data interchange. In *Heterogeneous Computing Workshop (HCW)*, 2000.

[10] Greg Eisenhauer, Beth (Plale) Schroeder, and Karsten Schwan. Dataexchange: High performance communication in Distributed Laboratories. *Parallel Computing*, 24:1713–1733, 1998. Elsevier, The Netherlands.

[11] P.Th. Eugster, P. Felber, R. Guerraoui, and A.-M. Kermarrec. The many faces of publish/subscribe.

[12] Geoffrey Fox and Shrideep Pallickara. An event service to support grid computational environments. *To appear in the Journal of Concurrency and Computation: Practice and Experience. Special Issue on Grid Computing Environments.*, 2002.

[13] Gnutella. gnutella.com. 2003.

[14] A. S. Grimshaw, W. A. Wulf, and the Legion Team. The legion vision of a worldwide virtual computer. *Communications of the ACM*, 40(1), 1997.

[15] Publish/Subscribe group at Brown University. Publish-subscribe.

[16] Dennis Heimbigner. Adapting publish/subscribe middleware to achieve gnutella-like functionality. In *Selected Areas in Cryptography*, pages 176–181, 2001.

[17] Marcos K.Aguilera, Robert E.Strom, Daniel C.Sturman, Mark Astley, and Tushar D.Chandra. Matching events in a content-based subscription system. In *Principles of Distributed Computing*, 1999.

[18] J. Feigenbaum M. Abadi and J. Kilian. On hiding information from an oracle. In *ACM Theory of Computing*, pages 195–203, May 1987.

[19] Benjamin Nguyen, Serge Abiteboul, Grégory Cobena, and Mihaí Preda. Monitoring XML data on the Web. *SIGMOD Record (ACM Special Interest Group on Management of Data)*, 30(2):437–448, 2001.

[20] C. Greg Plaxton, Rajmohan Rajaraman, and Andrea W. Richa. Accessing nearby copies of replicated objects in a distributed environment. In *ACM Symposium on Parallel Algorithms and Architectures*, pages 311–320, 1997.

[21] Antony Rowstron and Peter Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. *Lecture Notes in Computer Science*, 2218:329–??, 2001.

[22] Antony I. T. Rowstron, Anne-Marie Kermarrec, Miguel Castro, and Peter Druschel. SCRIBE: The design of a large-scale event notification infrastructure. In *Networked Group Communication*, pages 30–43, 2001.

[23] Aleksander Slominski, Y. Simmhan, A.L. Rossi, M. Farrellee, and D. Gannon. Xevents/xmessages: Application events and messaging framework for grid. Technical report, Indiana University, 2001.

[24] C. Wang, A. Carzaniga, D. Evans, and A. Wolf. Security issues and requirements in internet-scale publish-subscribe systems.

[25] B. Y. Zhao, J. D. Kubiatowicz, and A. D. Joseph. Tapestry: An infrastructure for fault-tolerant wide-area location and routing. Technical Report UCB/CSD-01-1141, UC Berkeley, April 2001.

[26] S. Zhuang, B. Zhao, A. Joseph, R. Katz, and J. Kubiatowicz. Bayeux: An architecture for scalable and fault-tolerant widearea data dissemination, 2001.

System	Category	System Architecture	Matching Algorithm	Multicasting Algorithm	Language Support
Gryphon	Content-based	Client-Server model	Matching tree	Link matching algorithm	Java-based
Scribe	Subject-based	Peer-to-peer model	Lookup table	Multicast tree	Java-based (FreePastry) and C-based (SimPastry/VisPastry)
Bayuex	Subject-based	Peer-to-peer model	Lookup table	Hashed-suffix mesh algorithm	Java-based
Siena	Content-based	Client-server model	Filtering algorithm (Binary Decision Diagrams)	Event notification service	Java-based and C++ based
Narada	Content-based	Client-server model	Matching tree algorithm	Link matching algorithm	Java-based
XMessage	Hybrid	Client-server model	lookup table and SQL like queries	Event notification service	Java-based
Echo	Hybrid	Peer-to-peer model	lookup table and filter function	Event notification service	Java, C++ and C

Table 1: Table for comparing different publish-subscribe systems

A Source Code for publisher

```
/**************************
 Independent study report
 summer 2002
 directed by Prof. Beth Plale
 @author By Ying Liu
**************************/
import siena.*;

class SimpleNotif implements Notifiable {
    Siena siena;

    public SimpleNotif(Siena s) {
siena = s;
    }

    public void notify(Notification e) {
System.out.println("local notifiable: " + e.toString());
 try {
     siena.unsubscribe(this);
 } catch (SienaException ex) {
     ex.printStackTrace();
 }
    }

    public void notify(Notification [] s) { }
}

//this class will generate three notifications
//and publish these notifications to its receiver server

public class InterestGenerator {
    public static void main(String[] args) {
try {
           int i;
           HierarchicalDispatcher siena;
    siena = new HierarchicalDispatcher();

    switch(args.length) {
    case 1: siena.setMaster(args[0]);
    case 0: break;
    default:
System.err.println("Usage: InterestGenerator [server-address]");
System.exit(1);
    }

    Filter f = new Filter();
    f.addConstraint("name", Op.ANY, "");

    siena.subscribe(f, new SimpleNotif(siena));

    try {
               for(i=0;i<10;i++){
Notification alert = new Notification();
alert.putAttribute("make", "Ford");
alert.putAttribute("model", "Focus");
```

```
alert.putAttribute("year", 2000);
alert.putAttribute("color", "red");
siena.publish(alert);
System.out.println("publishing " + alert.toString());

alert.clear();
alert.putAttribute("make", "Ford");
alert.putAttribute("model", "Tauras");
alert.putAttribute("year", 2001);
alert.putAttribute("color", "silver");
siena.publish(alert);
System.out.println("publishing " + alert.toString());

alert.clear();
alert.putAttribute("make", "Buick");
alert.putAttribute("model", "Century");
alert.putAttribute("year", 2000);
alert.putAttribute("color", "White");
siena.publish(alert);
System.out.println("publishing " + alert.toString());
              }
            } catch (SienaException ex) {
System.err.println("Siena error:" + ex.toString());
    }
    System.out.println("shutting down.");
    siena.shutdown();
} catch (Exception ex) {
    ex.printStackTrace();
    System.exit(1);
}
    };
}
```

B Source Code for Subscriber

```
/**************************
 Independent study report
 summer 2002
 directed by Prof. Beth Plale
 @author By Ying Liu
*************************/

import siena.*;

//this class will new a filter and subscribe to the siena server
//then it wait for 5 minutes to get notification which satisfies with that filter
//from the siena server

public class InterestReceiver implements Notifiable {
    public void notify(Notification e) {
        System.out.println("I just got this event:");
        System.out.println(e.toString());
    };
```

```
    public void notify(Notification [] s) {
    }

    public static void main(String[] args) {
if(args.length != 1) {
    System.err.println("Usage: InterestReceiver <server-address>");
    System.exit(1);
}

HierarchicalDispatcher siena;
try {
    siena = new HierarchicalDispatcher();
    siena.setMaster(args[0]);

    Filter f = new Filter();
    f.addConstraint("make", "Ford"); // name = "Antonio"
    f.addConstraint("year", Op.GT, 1999); // age > 18

    InterestReceiver party = new InterestReceiver();

    System.out.println("subscribing for " + f.toString());
    try {
siena.subscribe(f, party);
try {
    Thread.sleep(300000); // sleeps for five minutes
} catch (java.lang.InterruptedException ex) {
    System.out.println("interrupted");
}
System.out.println("unsubscribing");
siena.unsubscribe(f, party);
    } catch (SienaException ex) {
System.err.println("Siena error:" + ex.toString());
    }
    System.out.println("shutting down.");
    siena.shutdown();
    System.exit(0);
} catch (Exception ex) {
    ex.printStackTrace();
    System.exit(1);
}
    };
}
```