Internet and Grid Computing 2003 – Final Report

XML Filtering in Peer-to-peer Systems

12/10/2003

Honguk Woo (honguk@cs.utexas.edu)

Pilsung Kang (pilsungk@cs.utexas.edu)

Computer Sciences

University of Texas at Austin

1. Introduction

Internet-scale Information dissemination systems have been investigated to provide timely notifications for occurrences of relevant information on the Internet to a large set of users. The publish/subscribe model has gained much attention for this purpose, but currently it is yet limited in that most system implementations are based on typical keyword matching methods.

In this project, we have first investigated an automata based approach for matching XML document with user profiles, XFilter[3] which can extend information filter functionalities in terms of expressiveness of user profiles and matching effectiveness. Then we have explored a state of art technology, JXTA[4] to create a p2p networking environment where peers publish user profiles to collect and filter XML documents. We have leveraged these two technologies to build the XML filtering system in p2p networks in which peers in the networks are allowed to share XML documents and each peer can express its interest to filter unwanted XML documents out. This XML filtering system is useful in that it avoids message flooding in a distributed environment. A peer inserts XML documents in shared resource pool and only subscribed peers for the documents are notified dynamically in this environment. Our work is different from traditional publish/subscribe systems in that users' interest is described in XPath[9] to support a wide range of application requirements without limitation on subscription expressiveness.

Currently although we apply two post-engineering concepts of the notification such as full replications of matched documents and partial replications of matched contents, it would not be difficult to apply other application specific requirements on the top of our prototyping implementation due to our modular implementation.

The final project report is structured as follows. Section 2 summarizes the basics of publish/subscribe models which would give a picture of our system architecture. Section 3 describes a JXTA based p2p application, XShare which is developed to share XML documents in a common p2p fashion. This section includes the details on JXTA protocols and discovery services. Our technical review on XPath and implementation of XFilter are followed in section 4. We then conclude with interesting issues we may identify as future directions after showing experiment results of our system.

2. Publish/Subscribe Model

Recently the publish/subscribe model has been popular in many application domains due to rapid growth on on-line popularity and its efficiency on the integration process. As the web environment has spawn new business models based on on-line transactions and communities with a certain interest, advanced services for providing rapid notifications of certain events have been deployed in the form of information dissemination in many domains including stock quotes, financial news, transportation and so on. An agent handling information collection publishes such information, which will be delivered to a group of users at edge networks. The delivery process is to go through a publish/subscribe system and the system ensures the delivery of the notifying information to all the interested users. Recent deployment is mainly focusing on millions users around world, thus scalability is a key issue in the publish/subscribe model.

It is known that the publish/subscribe model is efficient for integrating applications in distributed environments because interactivities among the applications in general rely on their anonymity, delivering messages without any knowledge on destination of the messages. An information provider can fully leverage a publish/subscribe system by letting the system handle the delivery process without specification of how to send and who to receive. In the publish/subscribe model, an information provider and a group of users relying on the provider are called a publisher and subscribers respectively. The messages between the publisher and subscribers form events, which are well structured through the transformation process from collected

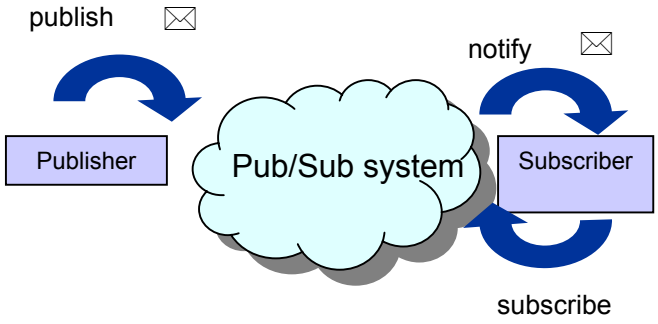
information to notification messages. Shortly in fully distributed environments, events are sent to whoever is interested in the events, whenever information providers want to share whatever they want.

The publish/subscribe model can be categorized in subject-based or content-based model.

- Subject-based publish/subscribe model[1]

Any event is tagged with subject that describes its content. Only users subscribing the subject of the event are delivered. This model can be said as group-based in which users make groups with the subjects and then multicast communication channels for the groups are supported. This model is inherently efficient and scalable because the mapping process from a particular event to a group of users who are supposed to receive the event should be taken care of only in the subject space with a fixed number. It supports, however, expressiveness in a limited way from users' perspective due to this fixed cases on the subjects. In a financial application domain, for example, a subject-based system may support such event labeling as "Dell" and other symbols, but users' interests in this domain are not fully described by using such categorization. Users may require correlation on symbols and even more details on contents such as quotes of related symbols or specific insiders' trading information.

- Content-based publish/subscribe model[1]



[Figure 2.1] Publish/Subscribe Model[1]

Identifying the tradeoff between efficiency and expressiveness, there are alternatives more focusing on utilization of contents in events in the matching process. While subject-based models can make easier to use simple matching algorithms by relying only subjects of events, content-based models need to develop more complex algorithms for filtering by more specific matching conditions. In the previous application domain, the "Dell" event can be more specific like "Dell with price > $10". This event definition is in general decided by users and thus ideal one would be in a natural language. While more expressiveness in filtering criteria is desirable in common settings, multiple dimensions on filtering criteria pose not only complexity issues of designing a sophisticated matching process but its scalability issue. Therefore most content-based models have relied on a typical scheme that implements matching mechanisms based on combinations of keywords and predicates over associative values of the keywords in that sense.

In this project, we have leveraged an efficient XML matching scheme, XFilter. Recent trends of XML have rapidly been increasing areas where XML data need to be analyzed for further processing such as information exchanges in business and business. Therefore, environments where XML data from multiple information providers are streamed and various users define their interest over such XML data are frequently referred to imply necessity of XML based publish/subscribe system architectures. In such environments both contents and structures are used to match XML data against users' interests, so a language for describing the interests requires to express desirable contents in specific structures. For this purpose, we use XPath, a W3C standard language. XPath can be used individually and also be a component of XML query languages such as Xquery[10] to address a certain part of XML data.

3. XML Sharing application in P2P

3.1 JXTA

Developed by Sun Microsystems, JXTA is an open source programming and computing platform to ease the development of P2P networking. JXTA protocols and components allow JXTA peers to perform the tasks that common p2p networks support[4]:

- Locate peers, peer groups, services and resources (discovery).
- Send messages to peers over virtual channels or pipes.
- Get status information on other peers
- Organize into peer groups.

JXTA is summarized in that "JXTA platform allows any connected device on the network ranging from cell phones and wireless PDAs to PCs and servers to communicate and collaborate in a P2P manner." The name comes from JuXTApose meaning "to place two entities side by side or in proximity."[4] The reason for

choosing this name is that the development team at Sun recognized that p2p solutions would always exist alongside the current client/server solutions rather than replacing them completely.

- Goals of JXTA[8]

Current p2p applications such as Napster, Gnutella, and ICO have similarities in terms of their functionalities: file sharing and instant messaging. One significant problem on current deployment of these applications is incompatibility between them. This problem results from no common development process and general platform for such p2p applications, leading each community of p2p applications to form closed one. Expecting p2p applications deployed continuously with more complex functionalities, one of major goals of developing JXTA is to provide a common language for p2p application developers, giving benefits to both development and integration process. By leveraging JXTA implementation, most fundamental parts of p2p applications such as resource management and communication can be easily component-based. Hence, the developers are allowed to focus on specific functions of the applications from the initial stage of their development process.

The goals of JXTA are summarized like followings:

- Interoperability: to support a wide range of distributed computing applications by developing a common set of general purpose p2p protocols
- Platform independence: any language, any OS, any hardware
- Ubiquity: to enable new applications to run on any device that has a digital heartbeat

JXTA provides multiple implementation bindings:

- Programming language: C/C++, Java
- System platform: Linux, Windows
- Network platform: TCP/IP, Bluetooth

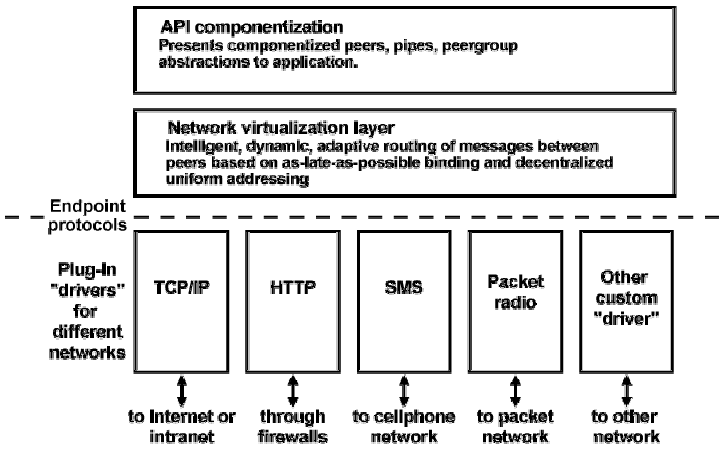
Very limited requirements are assumed on target devices in that any device with digital heartbeat can be JXTA node. This design concept of JXTA opens target applications to future areas like pervasive computing.

- Concepts of JXTA[8][4][7]
  - Peer: a peer is a JXTA node that implements JXTA protocols. A peer can publish services and resources available in its peer groups. It can be any device type with networking functionalities and JXTA protocol implementation: PC, PDA, server, mobile phone
  - Peer group: a peer group is a set of JXTA nodes who share a common interest. This peer group is the place where peers can dynamically join and leave. A peer should be associated with at least a single peer group and it can also be involved in multiple peer groups at the same time. The default peer group is "Net peer group" and if a newly joining peer can't find any peer group, then it may create the net peer group for itself. A peer group is a sort of organizational domain with peer group services for the peers in the group, seeking common goals. Rendezvous peers are responsible for discovery service and caching for advertisements, working as super-peers in cluster-based peer networks[2]. Relay peers route JXTA messages, and especially enable communication via firewall and NAT, the obstacles that incur network partitions. By this relaying mechanism, JXTA peers can communicate each other without partition problems and peer groups establish virtual regions for the peers in the groups.
  - UUID: UUID uniquely identifies a resource within the local run time environment. No global state is maintained, thus there is no guarantee of unique identifiers across the P2P network. Any resource type such peer, peer group, pipe, etc has its UUID.
  - Pipe: a pipe is a virtual asynchronous and unidirectional communication channel. It supports transparent fail over through dynamic binding. *Point to point pipes* connect exactly two pipe ends and *propagation pipes* connect one output to multiple inputs. Physical network deployment and implementation of the peers are encapsulated with peer endpoints, collections of network enabled address in peer nodes.
  - Advertisement: JXTA advertisements represent network resources such as service, peer, peer group, pipe etc. Peers publish their available resources by exchanging advertisements in XML formats. Since any resource should be discovered by JXTA services based on these advertisements, it is important to reduce communication messages created by the services. Hence, advertisements can be locally cached once discovered by peers and used during their lifetime specified by the time-to-live value. Even though it is preferred to minimise network traffics and to improve response times of resolving queries, advertisement cache is not mandatory because

JXTA does assume large range of peers from small devices such as nodes in harsh resource sensornets to servers with various capabilities. For such resource limited environments, advertisement caches may be beyond capabilities of the peers with small persistent storages.

- Peer Group Services[8][4][7]

JXTA implementation in Java provides peer group services, which are published in the advertisements of peer groups. These services consist of discovery, membership, access, pipe, resolver and monitoring services. It can be developed to meet specific requirements of applications, and generic implementation of the peer group services will apply unless replaced with the application implementations.



[Figure 3.1] JXTA Driver Structure[4]

- JXTA Protocols[8][4][7]

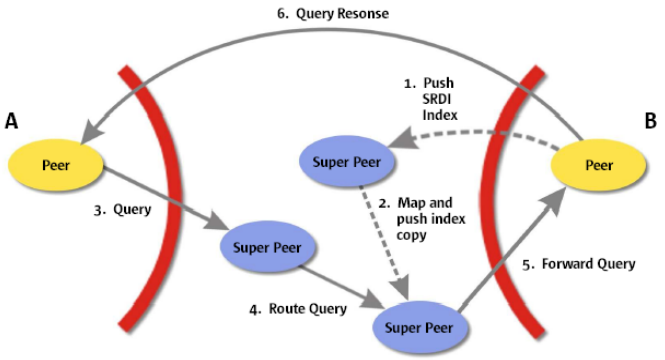
The JXTA specifies two protocol sets: core and standard. Core specification protocols include

- Endpoint routing protocol: used to dynamically find a route to send a message to another peer. If no direct route, then find the intermediary peers. It supports "plug-in" drivers for different networks
- Peer resolver protocol: used to send a generic resolver query to one or more peers, and receive a response (or multiple responses) to the query. Each query is addressed to a specific handler name, which defines the particular semantics of the query and its response.
- These core protocols must be implemented to be JXTA compliant, although these implementations do not guarantee interoperability of different peers. JXTA specification does not make assumption on network devices in terms of their transport protocol, so any message type on radio packet protocol or Internet-scale TCP can be supported by different JXTA binding implementations. Currently JXTA v2 from Sun includes TCP/IP, HTTP over TCP/IP and Bluetooth in its downloadable package. More implementation for small devices is expected in near future.

Standard service protocols include following four protocols:

- Rendezvous protocol: peers can subscribe or be a subscriber to a propagation service. Within a peer group, peers can be rendezvous peers, or peers that are listening to rendezvous peers.
- Peer discovery protocol: peers publish their own advertisements, and discover advertisements from other peers
- Peer information protocol: peers may obtain status information about other peers, such as state, uptime, traffic load, capabilities, etc.
- Pipe binding protocol: peers bind a pipe virtual connection to an actual peer endpoint

- JXTA Discovery[7]

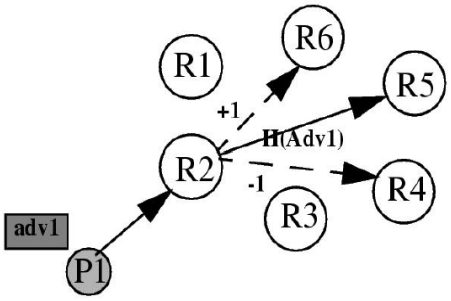


[Figure 3.2] Rendezvous Network[7]

Most improvement from JXTA v1 to v2 is from its discovery service implementation. In this project, we rely on JXTA discovery service and also refer to its scheme when designing our own protocol for broker cluster. JXTA discovery service is an asynchronous mechanism for discovering advertisements for peers, peer groups, pipes, services, etc. By utilizing this service, any peer can send query message to a specific peer and propagate the message to the JXTA network. In JXTA v2, resolution for the query message is processed by using

JXTA advertisements, the rendezvous super peer network, and shared indices of the network.

In the rendezvous super-peers network, queries are only propagated among rendezvous peers and edge peers only receive direct queries for their own advertisements. This



[Figure 3.3] DHT Replication on Neighbors[7]

new concept is applied in JXTA v2 to avoid message flooding. Any peer can be a rendezvous node if specified with its property and shipped with rendezvous protocol implementation. Edge peers publish indices of advertisement across rendezvous network using DHT(Distributed Hash Table). This index structure is called *share resource distributed index* in JXTA v2. There would be two possible approaches to find something in p2p networks: using some shared index over peers or relying on some network crawling mechanism.
These approaches have tradeoff between their maintenance cost and search cost. While the shared index structure is efficient for searching, its maintenance cost would increase significantly as its consistency requirement keeps tight, especially in highly dynamic environments. Contrary to the shared index, the crawling mechanism has no maintenance cost but its scalability is limited due to its inefficiency of searching. Rather than relying one of these, JXTA discovery leverages both from practical perspective, assuming a fluctuating and unpredictable network environment. Each rendezvous peer maintains an ordered list of known rendezvous peer in the peer group by their peer IDs. No strong consistency mechanism, however, is used to enforce the consistency of the list across all rendezvous peers. They periodically send and receive a list randomly chosen from their known rendezvous peers.

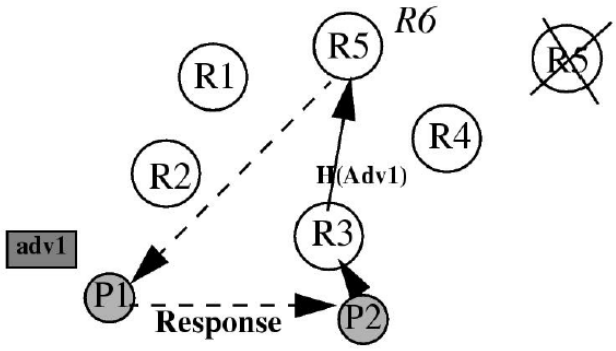
When R5, one of rendezvous peers in the figure 3.3, is the location for an index data (adv1) for the some network resource in P2 peer by the hashing result in DHT, the index is stored not only on R5 but also on its neighboring peers, say R4 and R6. These neighbors are chosen by a certain predefined number in the ordered list for rendezvous peers. Without changes on the rendezvous network, any query for the advertisement, adv1, is directly resolved by DHT.

If R5, the original location for the adv1, is down, then R3 in the figure 3.4 updates its ordered list of rendezvous peers. R3 still can resolve a query for the adv1 due to the neighboring replications. However the replication does not guarantee anything if it is partial, so more changes than the replication ranges require an alternative mechanism, *limited range walker*. If more changes occurred in the figure 3.5, the limited range walker would proceed both up and down way to find the adv1 in the peer group.

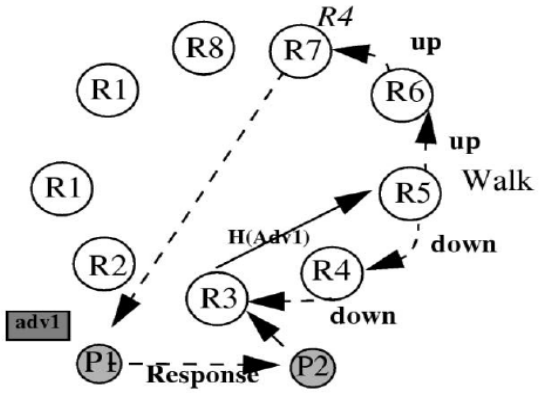
3.2 XShare Implementation

The JXTA based document sharing application with XML matching filters, XShare, has following functionalities:

- Join in Group: to join in a default NetPeerGroup. For rapid prototyping in this project, we allow only default peer group where peers are listed in our p2p network.



[Figure 3.4] Searching of Inconsistent DHT[7]

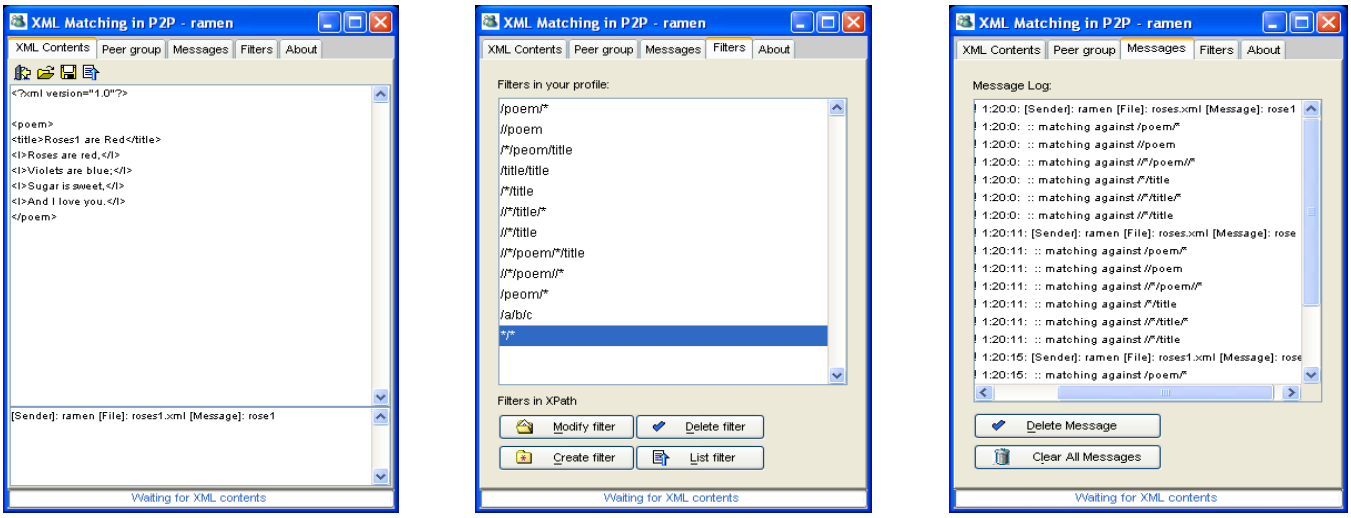


[Figure 3.5] Limited Range Walker[7]

- Share Document: to create metadata for a XML document which can be searched with its document title. This "share" process is that any XML document specified as users can be published as a form of the JXTA advertisement to let the document be shared in the default NetPeerGroup.
- Create Filter: to create XPath specification to filter XML documents out. In this project, a filter is edited in our test GUI window.
- Remove Filter (ongoing): to remove XPath specification in the p2p network.
- Activate Filter (ongoing): to allow the application to detect any documents shared in the NetPeerGroup and to be notified. Any matched XML document is automatically stored in the local directory for future usage, yet this replicate process is not an issue in this project.
- Deactivate Filter: to deactivate filters. When a filter is created, it is default to set to be activated.
- View Notifications: to browse a list of titles of shared documents and matching results against activated filters.

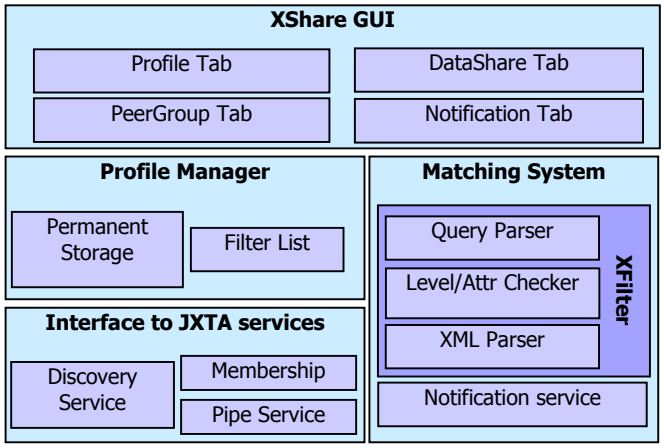
Our XShare prototype differs from general publish/subscribe systems in that a fixed set of what can be matched is not defined and its matching mechanism is more than a combination of keywords or predicates. The advantage of the prototype implementation results from its flexible XML matching mechanism using XFilter while this mechanism may be a burden to the underlying p2p network.

MyJXTA[4] is a sample application provided by the JXTA working group. It shows how to build a JXTA based p2p application in JAVA platform independent environments. Our work on XShare is based on such sample programs as MyJXTA and JxtaCast[4] from JXTA home, which provide us a starting point for GUI design and implementation of fundamental p2p functions. Our intent to design XShare is to provide a p2p application framework for testing the XML matching mechanism, XFilter in dynamic environments. We do not assume any specific application for target environments except large volumes of newly created XML documents, so in the project we focus mainly on implementing XFilter in p2p networks and designing a scalable XML matching architecture by leveraging JXTA technologies, especially in such dynamic environments. XShare in the figure 3.7 has four major components: JXTA based XShare peer, Profile manager, XFilter, and Notification service. In distributed deployments, every participant is a full-fledge XShare peer with all of XShare components, so a peer has subscriptions in XPath locally to process XML matching over dynamically generated XML documents. The local profile manager of the peer is responsible for handling subscriptions and filtering based on the subscriptions. In XShare, all subscriptions are encapsulated in *Filter objects* each of which is associated with a XPath string. Java JList and ArrayList are used to implement *Filter List structures, Filters* and instead of using a database engine for permanent storage, a text type file using FileWriter/Reader objects is maintained for the storage purpose.



[Figure 3.6] Screenshots of XShare

A peer can share XML documents by opening the documents and pushing them. The left window shows a XML data chosen. A peer can register subscriptions in XPath in the center window. The last window shows matching results between XML documents and XPath subscriptions



[Figure 3.7] Full-fledged XShare

XShare interacts with XFilter by implementing three major functions in the XFilter.Driver interface:

- *public ArrayList allMatch(ArrayList queryList, String xml_filename)*
  It attempts to match a file from XML stream against all of filters in ArrayList, which are handled by the local profile manager. Rather than evaluating an individual XPath over the file, XFilter constructs its query index structure of all possible XPath queries and returns a matching set of the XPath queries (unique IDs of the queries) to the local profile manager.
- *public void save_file(ArrayList queryList, String q_filename)*
- *public ArrayList open_file(String q_filename)*
  These two file interfaces are executed by a peer's local profile manager when the peer starts and terminates respectively.

4. XML Filtering Mechanism

In this section, we describe the basic data structures and algorithms used to implement an XML document matching technique, XFilter[3], of which the main algorithm was developed in UC Berkeley. We begin by presenting an overview of the XFilter mechanism consisting of the following components.

- Filter engine that performs matching operations between profile and XML documents
- Event-based parser for XML documents
- XPath parser for user profiles
- Post-matching process interfaces

The subscription language used in XShare p2p system is XPath, a language for addressing parts of an XML document that was designed to be used by both XSL Transformations (XSLT) and XPointer. With this user profile information, the XML document matching operations of XFilter are performed on every XML document that arrives at each XShare peer. Detailed information about XPath language can be found at http://www.w3.org/TR/XPath

4.1 XFilter

When an XML document arrives at the system, it is run through the event-based XML parser, which creates events that are responded to by handlers in the filtering process. Once the matching profiles have been identified for an XML document, the current implementation of our XFilter returns a MATCH (yes) and the document is kept by the XShare peer.

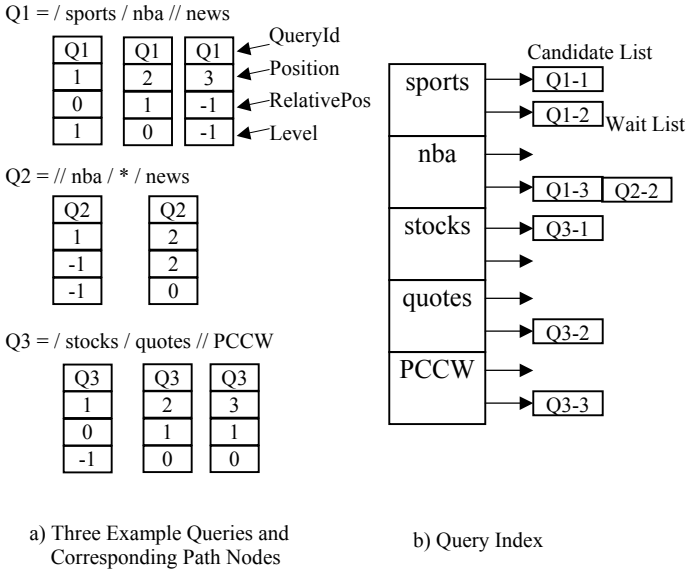
- Filter engine[3]

In order to achieve high performance in a large-scale environment such as the Internet, user profile grouping and indexing are implemented with an inverted index inside XFilter engine, called the Query Index, which is based on hash table. This Query Index is constructed by the decomposing process where each XPath query in the user profile is converted to a set of path nodes by the XPath parser. This process is equivalent to building the Query Index over states of a finite state machine(FSM). And then a XPath profile is a MATCH with a XML document once the FSM of the XPath query reaches its final state. Each of the decomposed path nodes serves as the key inside the Query Index hash table as well as the states of the FSM for the query. Path nodes are not generated for wildcard("*") nodes. When the XPath parser decomposes each XPath query in the user profile, it generates and stores the following information inside each path node[3].

  - *QueryId*: a unique identifier for the query to which this path node belongs.
  - *Position*: a sequence number, representing the location of this path node in the order of the path nodes for the query.
  - *RelativePos*: an integer value, representing the distance in document levels between this path node and the previous path node. It is set to 0 for the first node if the node does not contain a "Descendant" ('//') operator. It is set to -1 if a path node is separated from the previous one by a descendant operator. Otherwise, it is set to 1

plus the number of wildcard nodes between itself and its predecessor node.

- *Level*: an integer value, representing the level in the XML document at which this path node should be checked. If the *RelativePos* value of the path node is -1, then its *level* value is also set to -1. If the node is the first node of the query and has an absolute distance from the root, then the *level* value is set to 1 plus its distance from the root. Otherwise, the *level* value is set to 0. Note that this *level* value of a path node changes and is updated during the matching process.
- *NextPathNodeSet*: the pointer(s) to the next path node(s) of the query to be checked. In our current XFilter implementation, where we do not allow nested path expressions, each path node contains at most one pointer to the next path node.



[Figure 4.1] Path Node Decomposition and Content of the Query Index[3]

The figure 4.1(a) shows each of the decomposed path nodes of three example XPath queries and the corresponding information contained inside each path node. The Query Index is constructed using these path nodes, as shown in the figure 4.1(b). The Query Index hash table contains each element name of path nodes as the key, and two lists (Candidate List and Wait List) of path nodes as the value of the table. Each node in the Candidate List is the node of the query that is currently evaluated and represents the current state of the FSM. All the other path nodes that are evaluated in the future are placed in the Wait Lists of their respective element names. During the matching process, these nodes in the Wait Lists are copied and moved to their respective Candidate Lists, representing the state transition in the FSM of a query.

- SAX parser for XML documents

The SAX[12] parser in the Filter Engine parses an arriving XML document and generates parsing events when it sees a start tag, an end tag and data internal of an element node. And these events drive the matching process of user profiles against the document. The handlers of these events should be implemented such that the appropriate checking for the element, which generated the parsing event, against the current state of user XPath queries is done and the correct state transition is performed.

- *Start Element Handler*: when the SAX parser sees the start of an element tag, it generates the corresponding event. The Start Element Handler is called along with the following information: name, level, and attributes of the element tag. Then the handler finds the name of element in the Query Index hash table and performs two checks, level check and attribute filter check, on each path node in the Candidate List. The level check is to ensure that the level of appearing element in the document matches the expected level in the user query. If the path node contains a non-negative level value, in order for the check to succeed, it must be same as the level of the appearing element in the document. Otherwise, when the level value of the path node is -1, it implies that there is a descendant operator before this node and the level for the node is unrestricted. Then the check passes regardless of the element level. The attribute filter check is performed on any simple predicates that reference the attributes of the element. If both of these checks succeed, then the node passes. Now, if this is the final path node of the query, the query to which this path node belongs to reaches the final state and the document is a MATCH to the query. Otherwise, the state transition of the query happens by copying the next path node of the query from its Wait List to its corresponding Candidate List. During this transition, the level value of the copied node should be updated if its RelativePos value is not -1. The new level value

is set to the current document level plus its RelativePos value. If its RelativePos value is -1, we do not have to change it since it means the level value is also -1 and the level for the node is not restricted.

- *End Element Handler*: when the SAX parser sees the end of an element tag, it generates the corresponding event and this handler is called. In this handler, the corresponding path node is removed from the Candidate List, restoring the state of the list as it was when the corresponding start element tag was encountered.
- *Element Characters Handler*: when the SAX parser sees the data associated with an element, it generates the corresponding event and this handler is called along with the data passed to the handler. In our XFilter implementation, we use this data for a new XPath operator *ret()*.

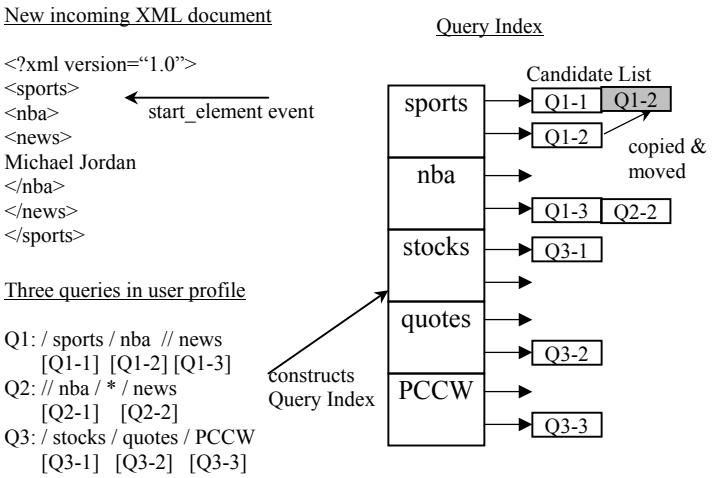
The figure 4.2 summarizes how the XFilter system works with an example. For the given user profile that consists of three XPath queries, the XPath parser decomposes them into individual path nodes, each of which contains information such as *QueryId*, *Position*, etc. Then the corresponding Query Index is constructed and initialized such that the path nodes, whose *Position* values are 1, are placed in the Candidate List and others are placed in the Wait List inside their corresponding elements nodes. Now, when a new XML document arrives, it is parsed by the SAX parser that generates events whenever the parser sees element nodes. And these events change the structure of the Query Index. For example, when the parser sees the <sports> node in the figure 4.2, it generates the 'start_element' event. According to this event, the Filter Engine searches the Query Index for the key 'sports' and performs *level check* and *attribute filter check* on the path nodes in the Candidate List, which is only 'Q1-1' in the figure. Since the level of the element node in the XML document, where the SAX event occurred, is same as the *level* value of the 'Q1-1' path node, the path node passes the check and the Filter Engine copies the next path node (Q1-2) and moves the copy from Wait List to the end of Candidate List of 'nba' index. While it copies and moves the path node, it also updates the *level* value of the copied node using the *RelativePos* value of the copied node and the current level or depth of the XML document. In the example, the copied node of 'Q1-2' will have a new *level* value of 2, which is the result of 1 (*RelativePos* value) + 1 (current level of XML document).
The SAX parser continues and then sees <nba> node in the document, finally generating the corresponding event. This process is repeated until the end of the document.

4.2 XFilter Implementation

We have implemented XFilter with J2SE v1.4.2.

- XPath parser

We have used Java Compiler Compiler, JavaCC[5] tool for implementing the XPath parser. Mostly the publicly available XPath grammar for JavaCC, which checks only if the given input is a valid XPath syntax or not is modified to add appropriate actions into this basic grammar to decompose a XPath query into a set of path nodes. From this modified grammar, we can get XPath parser Java source file for our purpose by running JavaCC on the grammar. Nested path expressions are not yet implemented in our prototype.



[Figure 4.2] XFilter Operations

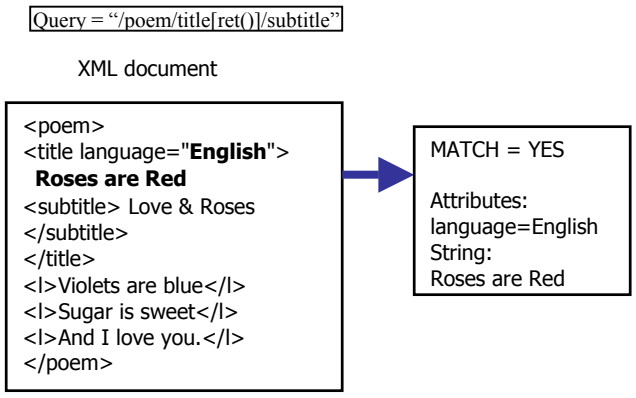
- SAX parser

We have used SAX v2 interface shipped in Sun Java package, and Apache Xerces[11] SAX parser implementation as the SAX parser. The followings are SAX 2 handler interfaces.

- *public void startElement (String uri, String localName,String qName,Attributes attributes)*
  This notifies the start of an element. The SAX parser will invoke this method at the beginning of every element in the XML document; there will be a corresponding *endElement* event for every *startElement* event (even when the element is empty). All of the element's contents will be reported in order, before the corresponding *endElement* event. It uses *uri* as the namespace of the XML document,

localName as the local name (without prefix), *qName* as the qualified name (with prefix), and *attributes* as the specified or defaulted attributes.

- *public void characters(char[] ch, int start, int length)*
  The parser will call this method to report each chunk of character data. The parser may return all contiguous character data in a single chunk, or it may split the returning data into several chunks; however, all of the characters in any single event must come from the same external entity so that the locator provides useful information. It uses XML documents character data from *ch* array, using *start* as the start position in the array and *length* as the number of characters to read from the array.



[Figure 4.3] *ret()* operator

- Filter engine : *ret()* operator

Currently, our implementation does only *level check* in Start Element Handler and *attribute filter check* is not yet implemented. Instead, in order to get information such as attributes and characters of a specific part of XML document when it matches user profile, XFilter is extended with our new operator *ret()* in XPath expression. With this operator, users of XShare peers can extract the only information they need within an XML document, as well as they can do filtering and matching the documents. The figure 4.3 shows an example of XPath query using *ret()* operator, and the output of the XFilter when a XML document matches the query. It returns all the information of the element node <title> in the document as well as the match result.

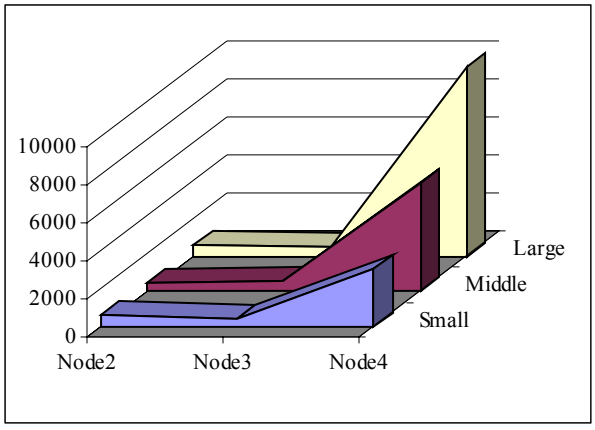
5. Experiments

In this section we show the test results of our prototype implementation. Our intent to conduct various test cases is two folds: evaluation of the group communication and the discovery service in JXTA v2 for J2SE, and evaluation of our XFilter implementation in terms of its scalability compared to a Java XPath engine, Jaxen.

We use four computers in the RTS lab of the University of Texas at Austin for most tests. All of them are Dell PCs and connected in a single LAN environment. Their specifications are followings: CPU 2.53Ghz Memory 512M, CPU 540Mhz Memory 256M, CPU 700Mhz Memory 256M, CPU 1.8Ghz Memory 1G. XShare prototype and test programs are all written in Java J2SE v1.4.2.

- Broadcast in a peer group

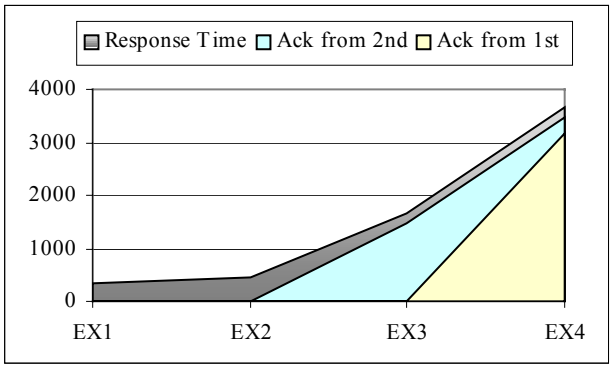
To evaluate the performance of broadcasting in a peer group, three XML documents with different sizes (small=150byte, middle=5k, large=50k) and varying peer groups are configured. We measure the collapsing time between the time just before sending a document from a sender to the group and the time when the sender receives all acknowledge messages from each peer of the group. The figure 5.1 shows that as the number of peers increased from two (node2) to four (node4), the time for broadcasting data, unit of which is millisecond, relatively increased when the group has four peers. As expected, in the node4 case, the data size also impacts the performance. Due to resource and time constraints, we are not able to test with more than four peers in the broadcasting test. We see, however, the JXTA propagation pipe implementation which is used in our broadcasting routine will even more affect the performance when dealing with heavy workloads like a large set of peers and transmissions of voluminous data.



[Figure 5.1] Broadcast time through Propagation pipes

- JXTA discovery service

This experiment is to examine how much the discovery service in JXTA is affected as the number of peers in a single peer group increases. The figure 5.2 shows the performance results of four test cases with various group settings. In all test cases, each peer sets to be TCP-enabled and multicast-enabled in its JXTA configuration. The first case EX1in which two XShare nodes are run in a single machine with different port setting, and the second case EX2 in which two XShare executing machines are connected, indicate the slight impact of applying multicast in the same subnet. Comparing the result of the EX2 case to ones of the EX3 and EX4 cases which have three and four peers respectively in a peer group, as expected, we observe the JXTA discovery service based on multicast in a single subnet is affected by peer group settings because of its underlying implementation. In fact the overhead from the multicast can be significantly alleviated by caching advertisements because a peer initiates a request for the discovery service to locate JXTA resources only when the information for the resource is not found in its local cache.



[Figure 5.2] Remote Discovery Time

- Matching mechanism

In real world scenarios, our XShare may be deployed as a middleware providing filtering services for large volumes of XML data in p2p networks. Therefore, to explore scalability of the matching mechanism in XShare, we compare our XFilter implementation to the current release of Jaxen, which has been used in various well-known projects. For this comparison, we send a 20K XML document to two different implementations of matching mechanism, XFilter-based and Jaxen-based ones with varying workloads. We use a Linux machine, lipton.cs.utexas.edu, in our department domain. This machine is equipped with two Xeon 1.8Ghz CPUs and 512M memory.

In the figure 5.3, the X-axis represents the number of XPath queries in the local profile manager and the Y-axis represents time in milliseconds. According to the test result comparing XFilter and Jaxen, we notice that XFilter scales up better than Jaxen. It is because XFilter applies multiple XPath queries simultaneously for a XML document in the matching process while Jaxen evaluates a pair of query and document individually. This result implies that in dynamic environments where commonly p2p applications are applied, utilizing parallelism in the matching process will be able to pave the way for reliable content-based publish/subscribe systems. It also implies the need for differentiating traditional database systems which have been focusing a large volume of data and efficient index architectures, and newly created p2p systems which have more dynamics on not only peers but data in the systems.

[Figure 5.3] XFilter vs Jaxen

6. Conclusions

We have implemented a p2p application which shares XML documents and matches a peer's interest against the shared documents. JXTA provides a network programming platform for p2p systems, by specifying and implementing a set of protocols which are independent of transport protocols, platforms, and development languages. In the project, we have used JXTA as a fast development toolkit for prototyping the XML matching application by building matching functionalities on the top of the interface implementation to JXTA v2 Java bindings. XFilter mechanism via which a sequence of XML documents over the query index structure is processed, has been implemented and tested for such matching functionalities in a p2p fashion.

Currently we identify two possible future works: a hybrid approach of XML-based publish/subscribe systems and a parallel model of XML matching mechanism. There is a tradeoff between subject-based publish/subscribe systems (efficient but less expressiveness) and content-based ones (expressiveness but hardly scalable)[1]. Since broker-based architectures[2] are promising for internet-scale notification services, a group of brokers and subscribers may share knowledge about what can be published by advertising XML schema of possible notifications. It would be also interesting to think of such environment where most XML documents published are very large and thus take too long to be analyzed. We envision a parallel version of XFilter to alleviate a large distribution of processing times for each pair of a huge document and XPath subscriptions.

7. References

[1] Design and Evaluation of a Wide-Area Event Notification Service, A. Carzaniga and A.L. Wolf, ACM Transactions on Computer Systems, 19(3):332-383, Aug 2001

[2] Designing a Super-Peer Network, B. Yang and H. Garcia-Molina. In Proc. 19th Int'l Conf. Data Engineering, Mar. 2003

[3] Efficient Filtering of XML Documents for Selective Dissemination of Information, Mehmet Altinel, and Michael J. Franklin, In Proceedings of VLDB 2000, Feb. 2000

[4] JXTA project home. http://www.jxta.org

[5] JavaCC: https://javacc.dev.java.net

[6] Jaxen: http://jaxen.org

[7] Project JXTA 2.0 Super-Peer Virtual Network: A Loosely-Consistent DHT Rendezvous walker

[8] Project JXTA v2.0. Java Programmer's guide

[9] XPath: http://www.w3.org/TR

[10] Xqeury: http://www.w3.org/TR/xquery

[11] Xerces: http://xml.apache.org/xerces-j

[12] SAX: http://www.saxproject.org