Boston College
Computer Science Department

Senior Thesis 2002

Michael Tierney

Middleman - Linking Data Sources & Wireless Devices

Prof. Edward Sciore

Introduction

The combined technological developments of Java and XML have helped create many advances in portability and customizable presentation of data. HTML, a cousin technology to XML, played an important role in the creation of, and shortly thereafter, the commercialization of the Internet. What HTML brought to the PC, XML will bring to cell phones, handheld computers, and PDA's. HTML allows for the formatting and presentation of text. XML takes an alternate approach to the handling of data. The markup tags in XML specify the type of data and characteristics about the data aside from presentation formatting. “HTML tells how data should look, but XML tells you what it means.”[1] The markup language says nothing about the presentation of the data; rather, it says what type of data is presented.

Within the document are tags created by the developer and placed around the data describing the data type. The XML document is then parsed and processed and formatting is determined by data type. This independence from formatting is one of XML’s major benefits. Because the documents do not specify explicitly how the data should appear, different presentations can be used, depending upon the user manipulating the data. For example, an animated, multicolor presentation with sound effects and voice narration would be one way of presenting data. This type of presentation would be ideal for a multimedia platform such as a powerful PC; however, a more limited cell phone browser, which lacks the capabilities for animation, sound, and even multiple colors, could not take full advantage of the data in this form. Instead, since XML supplies labels for the data, a designer could supply a schematic for the display of the data depending upon the capabilities of the device used to view it.

Furthermore, the document can contain a multitude of tags depending on the context of the data being presented. This is because developers can create their own tags based upon the needs of the application. In standard HTML, one would describe data as being placed in a specific location, in a certain font, color, and size. In XML, data is typed and described as FIRST NAME or STREET ADDRESS or STATE, or even something totally unrelated such as COLOR. The only limits to the possible tag values are the demands of the application and the imagination of the designer. This flexibility allows XML to be a great tool for transferring commercial data from databases to end-users. The querying system can pass both queries and results in XML format; portions of the query, such as the selection criteria can be labeled as such with XML tags, and the resulting data can be similarly identified, possibly with attributes such as table and field names, as needed. XML is extensible, in that it can grow to encompass and accommodate the data needs of the developer.

While these two properties of XML seem to present great advantages to a designer of a front end for database access, what is it about this markup language in particular that shows promise in place of some proprietary language? It is simply that XML is just that, a nonproprietary system. Multiple developers all over the world (since XML supports Unicode) can write applications that parse, process, and understand XML data. There is no Apple XML, no Sun XML, and (as of yet) no Microsoft version of XML. It is all one standard, as approved by the World Wide Web Consortium. If an enterprise uses applications from various software companies, attempting to integrate interoperable systems can become a nightmare of requisite data transformations. XML works as a "plain vanilla" or generic type of data, whereby all systems can communicate with each other. "A set of specifications and standards has been developed for XML transformations. These provide an intermediary layer that can act independently of application components to convert data from one format or style to another, leaving application components to perform business logic,"[2] explains McLaughlin in his book Java and XML. This middle-language allows for one standard in a world of innumerable data formats. If each format were converted to XML, and that markup document were used to transmit the data from system to system, each application would only have to convert to and from XML, not to and from the numerous different possible formats present in an enterprise's application pool.

My research, the development of the Middleman Server System, focused on combining the portable aspects of XML and Java with the seemingly natural pairing of XML and database connectivity. Taking advantage of XML's ability to function as a middle-language, my project allows for simple cross platform access to multiple database servers. Using a simplified SQL-like query language, Java's specialized API's for wireless and similar low computation devices, J2ME, and XML processors and parsers written in Java, the system translates a user's query, entered on a Java-compatible cell phone, into XML, and connects to an intermediary server, called a Middleman server. Middleman then examines the query, processes the commands, and performs a query on the back-end database. The Middleman server then does some preprocessing on the result set from the database, converting the results back into an XML format, and retransmits the resulting XML data to the user. The focus of the research was to develop the Middleman server, which accepts an XML query from an external client, transforms the query into the proper format for the target database, queries the database, and interprets, prunes, and personalizes the results. Also included in my research is the development of the client program, which establishes a connection between the client cell phone and my Middleman server and allows the user to interface with the databases through the server. Ideally, multiple clients could be developed to function on multiple platforms. Since the only communication between client and server is in standardized XML formatted messages, the Middleman Server System inherently supports cross-platform interaction. Using the known capabilities of the client device, the Middleman server can customize the result data to specifically suit the client.

By acting as an intermediary between the client and back-end database, my Middleman Server System decreases wasted bandwidth, decreases database processing, and allows for personalization and customization. It eases the processing load on both the database server and client by assuming a preprocessing role on both query and result transmissions.

Prior Work

There are currently many other papers and research efforts focusing on linking databases, data transport, WML, and wireless devices; however, many of these projects could benefit with the incorporation of XML as a data transport format and the Middleman Server System. These systems in general do not use XML, and function fairly well on their own without the technology; however, XML aids in the integration of these systems with other components by providing a standardized, nonproprietary data format.

In their paper “Extending the Data Services of Mobile Computers by External Data Lockers,” Villate, Pitoura, et al describe a service whereby a user stores data in an external data store and may access the data through a mobile computer. The system promises to “provide mechanisms that allow mobile users to use storage space external to their mobile computers by renting disk space in the intermediary element or the GSN.”[3] In effect, this system would be akin to ensuring access to data from any point via any device. However, in order for this cross-device feature to function, the data would need to be readable by multiple platforms.

Were the system proposed by Villate’s group to incorporate XML and act as a Middleman server as well as an external data locker, then a user could store data on the server in XML format and access the data, in parts or in whole, from a myriad of devices. Wireless phones, PDA’s, and personal computers could all access the same data stored in the same format.

In “SQL Server for Windows CE – A Database Engine for Mobile and Embedded Platforms,” the authors, two Microsoft employees, describe a separate version of SQL tailored specifically for the capabilities of mobile devices. This version of SQL is designed to deal with the storage, battery, and bandwidth restrictions imposed by wireless technology. I see no reason that multiple platforms would require multiple versions of SQL Server.

If the base framework for SQL requests and responses were XML correspondences, then a Middleman server could be used to sit between the client initiating the query and the SQL database. The server could take the incoming XML request, transform the query into proper SQL, and then process the results returned from the database system.

The database administrator institutes filters to ensure that a client receives the correct content, but does not specify the media used to convey that content. By allowing the database to ignore the issue of media type and passing that task off to the Middleman server, both the database and the client device can enforce “the abstractions of data access that application developers are familiar with.”[4] Back-end developers need not worry about the presentation of their data, and client-side programmers can rely on data being transmitted in a consistent format. Depending on what platform the client used to initiate the query, the Middleman server would tailor the response from the SQL database to match the capabilities of the client device. If a cell phone were used, the Middleman server would prune out graphical responses and pass on only textual interpretations of the data, while, if the device were a PC, graphics, audio, text, and animation would all be returned in the response from the Middleman database.

The client device, no matter what platform is in use, need only be able to send and receive XML documents – no explicit version of SQL is needed at the client side. The Middleman server must convert the XML from the client into a proper SQL query that the database system can process, and then interpret the database's response. SSCE, the ActiveX Control specified by Seshandri, "Can be performed over a wide variety of transports including wired and wireless LANs, wired and wireless modems, serial links, and infrared ports."[5] The transport ensures independence and the ability to connect to the database from many different platforms. What SSCE does for the physical layer of connection, XML does for the data being transferred. With Middleman, XML, and SSCE, both physical and logical independence are possible. Using XML and the Middleman server in this way could help eliminate the need for separate versions of SQL created to fit the needs of multiple platforms.

XML and Middleman could also extend the capabilities of the system described in Seydim, Dunham, and Kumar's paper, "Location Dependent Query Processing." In this system, the premise is that users would be able to access data relevant to their current locations using public terminals and wireless devices. The system uses a network of local servers that handle the information requests.

Because the user in this system is often in motion, the data returned by the query can be of various types. "MOD (Moving Object Database) queries have been classified as instantaneous, continuous and persistent query types."[6] That is, depending on the motion of the user and the item his or her query references, as well as the nature of the query, the database could return either one simple reply, a continuous stream for an unknown duration, or a persistent stream for a measured period of time. The Middleman server is ideally suited to the first type of query response, and can help ensure platform independence by converting the response of the server into an XML document. The two remaining query types provide an obstacle to the Middleman system, and the server must compensate for the continuous streams by dealing with discrete portions of the data, separating the stream into blocks of data, and converting the data pieces independently.

While this final condition limits the applicability of the Middleman system in a real-time continuous environment by forcing a discretization, the Middleman overall contributes interoperability to the system proposed by Kumar's group. It adds an additional layer to the scheme, and thus slows the process to a degree, but it also simplifies the tasks of both the client and the database. If the client device is a mobile system, such simplification can improve performance in terms of speed, memory usage, and battery power, thus making the Middleman server a valuable technology in this scheme.

Furthermore, Middleman can prune extraneous data and unusable formats, apply the correct DTD to the returned data, and, given a reference point, the system can query the correct local server. This last feature alleviates stress on the client device as well as on the local servers. The client need only pass its current location to the Middleman server, and the Middleman takes on the task of finding the nearest server offering the services the client demands. Middleman bears the brunt of the processing in this system and simplifies the tasks of the remaining components.

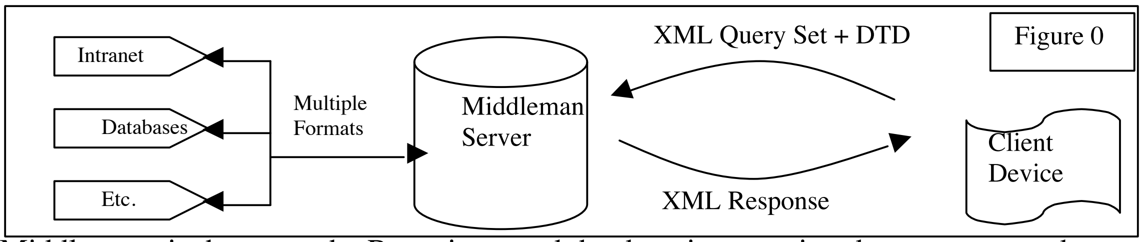
Another useful task suited to XML and Middleman is personalizing data for individual users. Subscribers to a given system could provide information requests asking for updates and furnish DTD's used in formatting the results according to the client's personal tastes. Data could be sent to a user's system either on demand or based upon a schedule of updates. "Integrated personalization and filtering are performed at each terminal,"[7] in the paper "Dynamic Personalization and Information Integration in Multi-Channel Data Dissemination Environments," by Goto and Kambayashi. In their paper they stipulate that, "Each passenger has a mobile terminal. There is a software agent having abilities to integrate and personalize information for the passenger in it."[8] The reasoning behind the author's decision to handle personalization on the client side is mainly in an effort to alleviate strain on the servers during high use periods.

The Middleman server system can help to alleviate server strain, simplify the task of the client device, and personalize data as well. If the reasoning behind the author's choice to force the client device to perform the personalization is sound, then the case where processing time becomes important is during high use periods. However, wireless bandwidth, which is often a major constraint, would also be taxed during these high usage periods. Therefore, forcing the client to filter and personalize requires that extraneous and often simply unusable data be transmitted and then immediately discarded by the client. This superfluous data transmission would entail very high bandwidth costs in an environment already depleted of available signals. The Middleman system provides a compromise that allows the constraint placed upon server processing time, as well as lessens the bandwidth costs of the system proposed in Goto and Kambayashi's paper.

First, the central and local servers would not have to waste processing time on filtering and customizing the information. They can simply pass the raw data and some basic information about the client device on to a network of Middleman servers. These servers can then apply the correct filtering depending on the client device. An additional possibility would be the storage of personalized DTD or Schema documentation on the Middleman server network that would allow for personalization of the data. After the data has been filtered and personalized, it can be forwarded on to the client device, thus reducing the amount of wireless bandwidth used. The mobile device need only append information regarding its capabilities to the original request, which is forwarded on to the Middleman system. While the adding of this supplementary information would slightly increase the bandwidth used in placing the information request, the benefits of prefiltering response data outweigh the costs imposed. The bandwidth used in sending a small text document is negligible when compared to that used when sending a color picture that the client device could not use.

Ozen's paper "Highly Personalized Information Delivery to Mobile Clients," deals with the customization of data on mobile devices. "The degree of personalization," he states, "becomes a key issue in such information services due to limited computation power of mobile devices and overwhelming number of potential of users with unrelated data."[9] The solution presented by Ozen and his colleagues is to create a system of profiles defined in XML-QL; each user would create a profile that listed all information services requested by the user and DTD's to apply to each service. Thus each user receives only the information requested in the profile, and that information is personalized and customized for the client.

The limitation placed upon this system is that the profile processor, which applies the profiles to incoming information streams, is based solely upon XML-QL, a query language that processes only XML documents. Ozen defends this limitation, stating that, "Since the queries will be executed on the documents fetched over the Internet, it is natural to expect the documents to be in XML, XML being the emerging standard for data exchange over the Internet."[10] However, with the addition of a Middleman server, the capabilities of Ozen's system can be expanded to deal with many more formats. Middleman processes data in the natural language of the data source, be it HTML, XML, or any of a variety of database response formats, and then converts that language into XML for transmission to the client. Middleman could improve the personalization system of Ozen's paper by sitting between the XML Repository and the data sources, or even by replacing the XML Repository and the Profile Processor altogether.



Middleman sits between the Repository and the data, intercepting data requests and responses and formatting the communications correctly. (See figure 0) Instead of requiring that all incoming data be in XML formatting, the Middleman server can handle multiple formats, yet still provide personalized data in a consistent format to the client device. Since the response to the client is sent in XML and applies a specified DTD, the client will receive a result containing only the information requested, in useable formats and media types.

The smartcard program proposed in "Toward Ubiquitous Database in Mobile Commerce" could also benefit from the application of XML technology and the Middleman server system. Again, by reducing the amount of information sent to the client device through filtering, personalization, and preprocessing, the computational and power demands on the smartcard can be greatly reduced. The goal behind Kuramitsu's system is to, "Place computers everywhere in the real world environment, providing ways for them to interconnect, talk, and work together."[11] Kuramitsu proposes a complex system of queries and objects that is finely tailored to work with the smartcard and greatly reduces the time needed to transmit the data. However, while this object/query system is fitted to the smartcard components, it fails to generalize well to other technologies. Other devices with greater computational power could deal with the additional overhead of a more universal format, such as XML. If the data is preprocessed and pruned by the Middleman servers, then the small overhead involved in the XML format is greatly reduced. While the tradeoff of interoperability versus speed does not favor XML and Middleman in this limited smartcard system, in more general cases the use of XML would be more favorable.

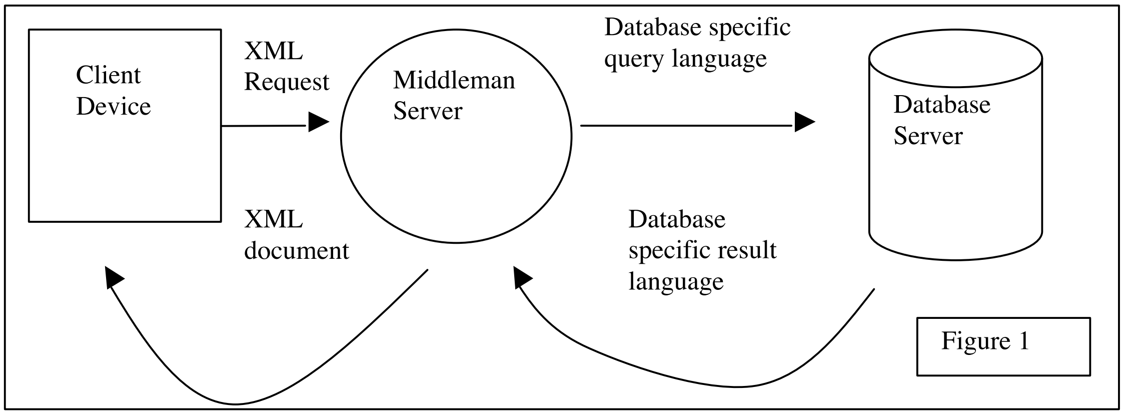
Another system that would benefit from the Middleman technology is described in the paper "Publish/Subscribe in a Mobile Environment" by Huang and Garcia-Molina. This article delves into the intricacies of a mobile subscription-based broadcast network consisting of three major components. "It consists of one or more Event Sources (ES), an Event Brokering System(EBS), and one or more Event Displayers(ED)."[12] The focus of the paper details the subscription management necessary for multiple mobile EBS servers to reduce network traffic and ensure delivery of the correct data to the correct customers. However, the basic system described in the paper could benefit from Middleman and XML. The EBS server(s) needs to broadcast a given piece of data to multiple heterogeneous users. These platforms used by these clients could vary from PC to PDA, and as such, a standardized data format that can be processed by any of a variety of systems, such as XML, would be ideal. Furthermore, since the data is routed to multiple platforms, preprocessing and prefiltering, done by the Middleman server system, greatly reduces wasted bandwidth and allows for further personalization of the data.

By placing the Middleman between the EBS and the ED, or even by integrating Middleman function into the EBS network, the Middleman can intercept signals bound for the ED and reformat the data. By applying either generic platform based or more specific user based transformations and conversions, the EBS/Middleman combination can deliver subscribed content to users in a consistent, well formatted, and useful manner. The EBS by itself forwards the data to the client device in the exact format sent out by the ES, but since the data is published by various sources, the formats can vary and may contain a variety of media. Middleman adds to EBS by standardizing the data format and customizing the media contained in the broadcasts.

Middleman, in combination with XML, has a great deal to offer the systems described in these papers. The major advantages my system provides is customization, standardization, and optimization. Customization allows the client to receive exactly the data requested and filters out unwanted data. Standardization reduces the programming and computation needs of the client devices. The only demand placed upon the device is the ability to process XML. Optimization functions by reducing the wasted bandwidth consumed by removing and filtering media types that client devices cannot interpret or display.

System Design

The basic premise of my system is to use a preprocessing server to intercept requests from a user device destined for a database and to perform intermediary processing tasks on both the request and the reply. This is done by first accepting a query request formatted as an XML document. The request is structured in a specialized Middleman query. The server then parses the request document to obtain key information, such as client identification, the requested database file, the operations to be performed upon that database, and the desired result format. The Middleman server then matches the client device type, established by examining the identification against a local XML file containing a device registry. Client capabilities, such as multimedia abilities, are listed in this registry and associated with the identification tag in the XML query. This last ingredient helps provide the major benefit of this system; by obtaining specific attributes pertaining to the computing device, be it a web-enabled cell phone, an internet-ready PDA, a desktop computer, or a BlackBerry RIM email device, the Middleman server can prune down the results and tailor the information returned to the device to best match its abilities.



The above diagram (figure 1) shows the lifespan of a query, generated at the client device, processed by the Middleman server, and passed on to the database. The result of the query is passed back through the Middleman, additional processing is done, and the adapted result is then passed back to the client.

The Middleman server sits as a layer between the client and the database system. It waits for an incoming connection until the client sends a connection request. Upon connection, the client sends a short XML document containing a customized, simplified version of a SQL query (see figure 2) and additional information. This query is then processed, and the middleman server generates a new query, based on the language required by the target database server.

```
<QUERY>
        <ClientID> Cell1</ClientID>
        <TARGETDB> addressbook </TARGETDB>
        <FieldList>
                <NumberFields> 3 </NumberFields>
                <Field1> LastName </Field1>
                <Field2> FirstName </Field2>
                <Field3> PhoneNumber </Field3>
        </FieldList>
         <SelectConditions>
                <NumberConditions> 1 </NumberConditions>
                <Condition1>
                        <Condition1Field> LastName </Condition1Suffix>
                        <Condition1Equality> StartsWith </Condition1Equality>
                        <Condition1Value> "H" </Condition1Value>
                </Condition1>
        </SelectConditions>
        .
        .
</QUERY>
```

Figure 2 – sample query (XML format)

An XML based query, while quite verbose, is unambiguous and very simple to parse. The middleman server can quickly determine which database system should be queried and what the structure of the query would be. For example, if the addressbook table referred to in the above figure resides on a database system which processes SQL queries, the middleman would convert the query into "SELECT LastName, FirstName, PhoneNumber FROM addressbook WHERE LastName LIKE = 'H%' and transmit that query on to the database.

Shortly thereafter the middleman server receives a reply from the database including the results of the query. This result document then undergoes a transformation into XML format, and the middleman then takes into account the capabilities of the client device. If the result set includes media beyond the capabilities of the device, such as frames, flash animation, graphics, or video, the middleman server attempts to present the content in an alternate format. To do this, the result set from the backend database must include multiple formats in the transmission to the Middleman server. From this large set of results, the Middleman Server System creates a subset that meets the capabilities of the requesting client. Pictures become words, charts and graphs become lists and statistics, etc. Personal computers could support such graphical media, and thus if a PC instituted the query the result would include multimedia content. Other devices, however, cannot.

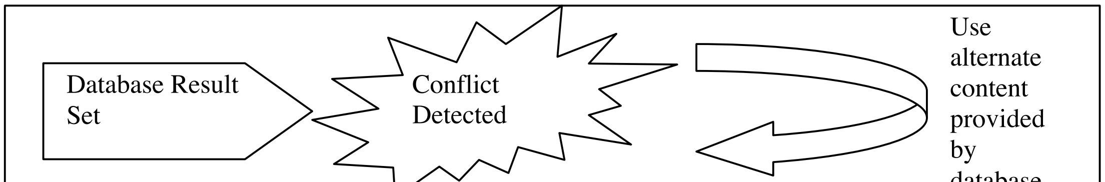


Figure 3

Finally, once the middleman server has determined that the result set from the original query contains only media displayable by the client device, the results are converted into an XML document, and that document is passed back to the client originating the query, along with any external files, such as sound files, images, or other file types.

The multitude of platforms this system attempts to accommodate requires many versions of the client program. A PC client would be quite different from a basic PDA client, but all the programs serve similar purposes and share key components. The user input must be collected together into an XML document, and a connection to the middleman server must be established. Then the device must transmit the query document, receive the results, and display the result of the query in the appropriate manner. Since the client is written in Java, the core classes can be reused in various client versions. The few classes that deal specifically with the device itself can be altered, but the essential code remains the same.

The following are two sample use cases for the Middleman Server System, detailing the tasks of the client device, the server, and the backend database.

Use Case I – Initial connection & configuration of back-end database/Middleman communication.

1) Using a web interface to the Middleman system, a user elects to add a database to the subscribed database listing.
2) Middleman responds by displaying a data entry screen, prompting the user for the database name, the field titles, and the address of the database, as well as any security restrictions necessary.
3) Middleman sends a test query to the database to ensure that all fields entered by the user exist in the database, and that the database as a whole exists in the location specified. If this test succeeds, Middleman appends the database on the local listing of subscribed databases. Middleman does not, at the present time, make any assumptions as to the data types returned in the various fields from the database. Middleman knows only the number and names of the fields in the database and the permissions required for access.

Use Case II – Initial connection & configuration of client/Middleman communication.

1) Client performs an initial connection to the server system, broadcasting a request for membership with the system. This request includes the type of device (PC, PDA, Cell Phone, etc) the client is using to connect.

2) Middleman server system receives the request from the client, adds the device to a listing of subscribed clients, and assigns the client a unique identification code. This code serves to allow the Middleman Server System a simple method of identifying the initiator of any queries. Using the identification code the system can access the database of clients and determine the device type used in initiating the query, and thus tailor the response to suit the capabilities of the client. The system responds to the initial request with an XML document containing the assigned client identification. Using the "CurrentCount" field of the registry, as shown in figure 4, the server would assign a designation of "JavaCell4" to the client, and update the registry file accordingly.

```
<Registry>
<Java_Cell_Phones>
        <CurrentCount>4</CurrentCount>
        <Subscribers>
                <Client>JavaCell1</Client1>
                <Client2>JavaCell2</Client2>
                <Client3>JavaCell3</Client3>
        </Subscribers>
        <AbilityList>
                <MIDISound>True</<MIDISound>
                <WAVSound>False</WAVSound>
                <PNGraphics>True</PNGraphics>
                <BMPGraphics>False</BMPGraphics>
                <AdvancedGraphics>False</AdvancedGraphics>
                …
        </AbilityList>
        …
</Registry>
```

Figure 4

3) If the client provided a device identification code not recognized by the system, the Middleman Server System uses a default category containing a limited ability list.

4) The client device receives the identification code from the Middleman system and saves the code in a local configuration file. This code will be used in all future correspondences between the client device and the Middleman.

Use Case III – The client, having already initialized the device with the system, performs a query on a database through the Middleman Server System.

1) The client then attempts to initiate a query on the database system. A transmission is sent to the Middleman system, requesting a listing of available databases and their fields. This is done automatically when the client begins the query process.
2) The Middleman system receives the request from the client and responds with an XML document containing the most recent listing of subscribed databases and the fields of each database.
3) Upon receipt of the database listing from the server, the client parses the XML document and extracts a listing of the databases. The client program displays a menu of the available databases.
4) The user selects the target database from the listing provided by the client program. The client then extracts a listing of the fields belonging to the database from the XML document provided by the Middleman server. From this list the user selects fields which the Middleman server will use as a projection to query the database.
5) After prompting the user for the projection list, the client program displays a screen whereby the user can add selection conditions to the query. The user can

enter the conditions by selecting fields. The client then prompts the user for the type of equality (starts with, equals, less than, greater than, etc) and the value.

6) After obtaining the projection and selection conditions, the client program assembles an XML query, including the conditions, the target database, and the client identification (obtained in the previous use case) and transmits the query to the Middleman system.
7) The Middleman server then parses the XML request from the client and extracts the necessary information. First it identifies the target database from within the query. Then the system matches the target against a local document which acts as a mapping from database names to the fields as well as the query language of the database.
8) Middleman then begins to construct a native language query depending on the result of the comparison in step 7. It transforms the remainder of the original XML request into a SQL, XML-QL, or other language type query. It also validates the fields specified in the query with the local file enumerating the fields of each database.
9) After constructing the query, the Middleman server transmits the request to the appropriate backend database and waits for a response.
10) Upon receiving a response from the database, the Middleman server matches the response to the original query. It then compares the identification field in the original query to a registry of devices and determines the device type originating the query.

11) According to the device type specified in the registry, the Middleman server then prunes the response received from the backend database server. For example, if a Java cell phone originated the request, the identification specified in the original query would map to a cell phone in the registry. The Middleman server has a listing of the capabilities of each type of device, and a default low-capability setting for devices not matching any entry in the registry. Using the listing for a cell phone, the Middleman server would know to remove any non-png (Portable Network Graphic) files, any audio files, and any video files.
12) In the case that a conflict is found between the content presented by the database and the capabilities of the client device, Middleman elects to either remove the content, of if possible, present the information in an alternate format. Thus, it is the responsibility of the backend database to provide a low-tech representation of high-level data when possible. In this example, it would be the responsibility of the database to provide images in png format in addition to any more complex formats used, thus allowing Middleman to select from a variety of data types.
13) Middleman does not have any a priori knowledge of the data types of the database fields, thus maximizing the plug-and-play ability of the interface. The back-end database could change completely, and the only update needed on the Middleman side is a new listing of field names. Middleman must analyze the results as they are returned and discover the data type itself. By pushing the responsibility for determining the data type of the result set onto the Middleman server compatibility issues are reduced and the bulk of the processing load remains Middleman's responsibility.

14) The Middleman system then converts the pruned down result set into an XML document and passes it on to the originating client. Included in the XML document is a listing of the fields requested and the data type of the returned fields.

15) The client receives the XML document and parses it, extracting the results and the data types. Using a DTD, which is provided as a default in the client, but can also be updated either directly by the user or indirectly through an update from the Middleman system, the client displays the results for the user. The user can then elect to either clear the result and initiate a new query or save the results locally.

These use cases represent typical scenarios faced by the Middleman Server System. The system entails a large number of connections between the client and Middleman servers, but attempts to minimize the bandwidth used in each transmission. While this design does force the backend database to transmit more data in a standard reply in order to meet the demands of the client device, it also allows the database to focus on data retrieval tasks, and assumes all filtering and post processing tasks. Thus, despite the additional bandwidth used between the Middleman system and the backend databases as well as the need for additional connections, my system reduces the wireless bandwidth used between the Middleman Server System and the client device and also simplifies the IO tasks of the databases.

Conclusion

The Middleman server system acts as an interpreting and filtering system providing customization, standardization, and optimization to a variety of client devices. The purpose of the system is to allow a multitude of platforms to access data sources without concern for formatting and media type. The Middleman can also act in concert with other systems specialized for use with mobile devices. The Middleman system provides a bridge between these specialized services and the clients. It generalizes the data and allows XML to truly be the standard in data communication.

As shown in the table in figure 5 below, the Middleman system effectively reduces the bandwidth used in response to user queries. Using a series of queries on the addressbook database requesting fields containing large data types, such as images, I established a baseline average bandwidth usage over the series. Similarly, by running fifteen queries through the Middleman Server System and allowing the server to prune the results, I established a basis for comparison between the original system and Middleman. By eliminating useless media and extraneous overhead, the Middleman system cuts down on wasted power, time, and processing. In the example used, the direct connection had no a priori knowledge of what types of data would be useable by the client device. Therefore, when the client requested a projection on the photo field, the database returned all the formats in that field. However, only the png format was useable by the client device. The bitmap and jpg files sent by the database were discarded as incompatible. While Middleman entailed more connections and additional overhead, the actual bandwidth used was greatly reduced, mainly because the server was able to strip the response of the bmp and jpg files that it knew the client could not use. While the Middleman system requires a greater number of connections, due to the need to pass additional database information, the bandwidth saved in the average case was significantly lower, and contained a great deal less unusable information than direct database access.

	Number of Connections	Bandwidth Used
Direct Connection	2	420,467
Using Middleman	5	89,116

Figure 5

If the standard database connection provided adequate content to meet all demands of the multiple client types supported by Middleman, the bandwidth usage greatly increases.

By converting various data types into XML format, filtering the data, and eliminating extraneous media, the Middleman server links data sources and wireless devices in a seamless manner. The use of XML as opposed to a proprietary alternative assures that the data will be accessible by all, easily processed, logically arranged, and free of irrelevant formatting data. By describing the data by type information instead of formatting, XML presents a document that is easy to process, convert, and manipulate. XML and Middleman function to make data of many types and from many sources easily accessible to users at home, at work, and on the road.

Future Work

At the present time, the majority of Middleman configuration and updating is done through directly manipulating configuration files on the only implemented server. Thus, in order to add service, databases, or any other components, direct editing of the XML configuration files must be performed. Also, the current server is the only machine in the system, and thus does not provide adequate coverage for a multitude of clients in numerous locations. Future efforts can expand the system and also simplify and automate the configuration process.

Ideally, the Middleman Server I have implemented would be one of many in a network of servers. These servers would periodically update each other with information on subscribed client devices, identification codes, and member databases. Therefore, multiple clients could connect to the servers from many locations, access the servers, and query databases. By adding servers, and placing them close in terms of network topology to the backend databases, the system can minimize the amount of high bandwidth usage in communication between the databases and the Middleman servers.

Additionally, client programs for other devices need to be implemented, as do interfaces with more database types. The Middleman server itself needs a web-based interface for manually adding database server information, new DTD's, and new additions to the device capability records. Each user should be able to create a user profile on the server system, associate all his or her client identifications with that profile, and create custom DTD's and standard queries. Then, when connecting using new devices, these DTD's can be uploaded to the device. Also, a user's frequently used queries can be saved in the profile on the Middleman server, and then quickly accessed via a separate menu. This listing of queries would improve the speed and simplicity of usage of the client program and would also reduce the amount of information that would have to be transferred from the client to the Middleman server to initiate the query. Instead of a verbose XML query document, the client would only have to pass a reference to the index number of the stored query, and the Middleman could rapidly implement the query. Instead of converting an XML query into the native language query of the appropriate database server, the Middleman server would already have the converted query stored locally, and would use that cached query on the backend database.

With the extension of the Middleman Server System in these ways, the system would be a viable product in the ever-growing attempt to deliver user based personalization of content, balanced by the need to minimize bandwidth use. Middleman presents an effective tool for corporations as well as individual users, allowing employees to access essential data simply and quickly.

Appendix – Limited Source Code Listing

```
import java.util.Vector;

public interface GenericDatabase
{
   public Res getNextResult();

   private void makeQuery(String targetdb, Vector projects, Vector whereconds)


}


import javax.microedition.lcdui.*;

public interface Controller
{
            public ExternalDB getExternalDB();
            public Displayable currentScreen();
            public void nextScreen(Displayable display);
            public void lastScreen();
}

import java.util.*;

// the immediate results of the query are stored here

public class Datum

{
   private Stack current_ns = null;
   private Vector fieldnames = null;
   private Vector rawinput = null;
   private Vector adjustedinput = null;
   private int number_fields = 0;

   public Datum()
   {
      fieldnames = new Vector(5,5);
      adjustedinput = new Vector(5,5);
      rawinput = new Vector(5,5);
      number_fields = 0;
   }

   public void setNumberFields(int n) {
                  number_fields = n;
            }

            public int getNumberFields(int n) {
                  return (number_fields);
            }

            public void processData(){
```

```
	// this method takes the data stored in the rawinput vector and splits it up into fully defined field 
names
	// and "adjusted input" which is a vector of the actual data corresponding by index to the field 
names

			boolean go = true;
			Stack hold = new Stack();
			int count = 0;
			current = 0;
			while(go)
				{
				String temp = rawinput.get(count);
				if( (temp[0] == '<') && (temp[1] == '/'))
					{
					if(current_ns.empty())
						system.out.println("Stack error!  Malformed XML or improper 
stack usage resulted in poping an empty stack");
					else
						{
						current_ns.pop();
						current--;
						}
					}
				else if( (temp[0] == '<') && (temp[1] != '/'))
						{
						current_ns.push( (Object) (temp) );
						current++;
					}
					else {
						String builder = new String();
						for(int i = 0; i < current; i++)
							hold.push(current_ns.pop());
						for(int i = 0; i < current; i++)
							{
							String t = ( (String)(hold.pop());
							builder.concat(t); // builder now holds the 
'namespace' of the value that we have reached
							current_ns.push((Object)(t));
						}
						fieldnames.setElementAt( ((Object)(t)), count);
						// now that we have captured the label for the value, we'll grab 
the value and put it in the adjustedinput
						adjustedinput.setElementAt( ((Object)(temp)), count);
				count++;  // onward!
				if(count > rawinput.size())
					go = false;
			}
			if(!current_ns.empty())
				system.out.println("Stack error!  Malformed XML or improper stack usage 
resulted in a nonempty stack at conclusion.");

	}

	public void addInput(String s){
	// this method takes a string of raw xml input and stores it in the rawinput vector, to be later 
processed by the
```

```
		// processData method

			rawinput.add( (Object) (s) );

		}

		public String getField ( int n) {
			// this method returns the fully defined name of the indexed field (ie
"<phone_number><home_phone><area_code>"
			String s = new String((String)(fieldnames.get(n)));
			return(s);
			}

		public String getResultVal ( int n) {
			String s = new String((String)(adjustedinput.get(n)));
			return(s);
			// this method returns the value of the indexed field (ie "617")
			}


}

import java.util.*;
import java.io.*;
import org.kxml.*;
import org.kxml.kdom.*;
import org.kxml.io.*;
import org.kxml.parser.*;


public class dbInfoLoader{

public dbInfoLoader(){
		currentIndex = 0;
		databaseNames = new Vector(5, 1);
		numberOfDatabases = 0;
		databaseFields = new Vector(5, 1);
		numberOfFields = 0;

}

public dbInfoLoader(String newPath){
		currentIndex = 0;
		databaseNames = new Vector(5, 1);
		numberOfDatabases = 0;
		databaseFields = new Vector(5, 1);
		numberOfFields = 0;
		fileName = newPath;

}

public int getNumberOfDatabases(){

		return numberOfDatabases;
}
```

```
public int loadDatabaseList() throws IOException{

        /* xml format
                        <dblist>
                                <NumberDB>1</NumberDB>
                                <DB0Name>AddressBook</DB0Name>
                                        <numberFields>10</numberFields>
                                        <fields>
                                                <field0>FirstName</field0>
                                                <field1>MiddleInitial</field1>
                                                <field2>LastName</field2>
                                                <field3>HomePhone</field3>
                                                <field4>HomeStreet</field4>
                                                <field5>HomeCity</field5>
                                                <field6>HomeState</field6>
                                                <field7>HomeZip</field7>
                                                <field8>MobilePhone</field8>
                                                <field9>Photo</field9>
                                        </fields>
                                </AddressBook>
                        </dblist>
                */

        try{
                InputStream is = Connector.openInputStream(fileName);
                InputStreamReader isr= new InputStreamReader(is);

        }
        catch(Exception e){
                return -1;
        }
        XmlParser xp = new XmlParser(isr);
        Document doc = new Document();
        doc.parse(xp);
  Element dbElement = document.getElement("dblist");
  String numDBs = getTextFromElement(dbElement, "NumberDB");
  numberOfDatabases = Integer.parseInt(numDBs);
  for(int i = 0; i < numberOfDatabases; i++)
        {
                String c = new String("DB");
                c = c.concat(temp.toString());
                c = c.concat("Name");
                String temp = getTextFromElement(dbElement, c);
                databaseNames.add( (Object)temp);
        }
}

public int getCurrentDatabaseIndex(){

        return currentIndex;
}

public void useDatabase(int index){
        currentIndex = index;
}
```

```
public Vector getDatabaseNames(){
        return databaseNames;
}


public Vector getFieldNames(){
        /* xml format
                        <dblist>
                                <NumberDB>1</NumberDB>
                                <DB0Name>AddressBook</DB0Name>
                                        <numberFields>10</numberFields>
                                        <fields0>
                                                <field0>FirstName</field0>
                                                <field1>MiddleInitial</field1>
                                                <field2>LastName</field2>
                                                <field3>HomePhone</field3>
                                                <field4>HomeStreet</field4>
                                                <field5>HomeCity</field5>
                                                <field6>HomeState</field6>
                                                <field7>HomeZip</field7>
                                                <field8>MobilePhone</field8>
                                                <field9>Photo</field9>
                                        </fields>
                                </AddressBook>
                        </dblist>
                */
        try{
                InputStream is = Connector.openInputStream(fileName);
                InputStreamReader isr= new InputStreamReader(is);

        }
        catch(Exception e){
                return null;
        }
        XmlParser xp = new XmlParser(isr);
        Document doc = new Document();
        doc.parse(xp);

   String dbIndex = "DB";
   dbIndex.concat(currentIndex.toString());
   dbIndex.concat("Name");
   Element dbElement = doc.getElement(dbIndex);
   /*Element correctDb = dbElement.getElement(dbIndex);
   String numDBs = getTextFromElement(dbElement, "NumberDB");*/
   String numFields = getTextFromElement(dbIndex, "numberFields");
   numberOfFields = Integer.parseInt(numFields);
   Element el = doc.getElement("field"+currentIndex);
   for(int count = 0; count < numberOfFields; count++)
                {
                String fieldN = getTextFromElement(el, "field"+count);
                databaseFields.add( (Object)fieldN);
        }
        return(databaseFields);
}

public Vector getFieldNames(int d){
```

```
            this.loadDatabase(d);
            return(this.getFieldNames);
}

public int getNumberOfFields(){

            return numberOfFields;
}


protected String fileName = "/midp/thesis/dbinfo.xml";
protected int currentIndex;
protected Vector databaseNames;
protected int numberOfDatabases = 0;
protected Vector databaseFields;
protected int numberOfFields = 0;
}


import javax.microedition.rms.*;
import java.io.DataOutputStream;
import java.io.ByteArrayOutputStream;
import java.io.IOException;
import java.io.ByteArrayInputStream;
import java.io.DataInputStream;
import java.io.EOFException;
import java.util.*;

 public class ExternalDB implements RecordFilter, GenericDatabase {

   private RecordStore recordStore = null;
   public static String symbolFilter = null;


            public ExternalDB()
               { // this is only if the database response is cached, which I don't cover
                 // so this constructor shouldn't do much of anything
               Res compile = null;
               }

   public Res getNextResult() {

      ByteArrayInputStream bais = new ByteArrayInputStream(candidate);
      DataInputStream inputStream = new DataInputStream(bais);
      String name[MAXFIELDS] = null;
                  compile = new Res();


      try {

         Datum dpiece = DataStore.readDatum(inputStream);
         int numfields = dpiece.getNumberFields();
         compile.setNumF(numfields);
         while (int n = 0; n < compile.getNumF(); n++)
           {
                                          compile.addField(n, dpiece.getField(n));
```

```
                                              compile.addValue(n, dpiece.getResultVal(n));
                                }
        }
        catch (EOFException eofe) {
            System.out.println(eofe);
            eofe.printStackTrace();
        }
        catch (IOException eofe) {
            System.out.println(eofe);
            eofe.printStackTrace();
        }
        return (compile);
    }



    private void makeQuery(String targetdb, Vector projects, Vector whereconds)
    {
                        // whereconds is a vector that holds an alternating series of matching (field, value) strings
        Querytype q = new Querytype;
        ByteArrayOutputStream baos = new ByteArrayOutputStream();
        DataOutputStream outputStream = new DataOutputStream(baos);
        String temp1, temp2;
        try {
            q.setTarget(targetdb);
                                q.setNumFields(projects.size());
                                for(int n = 0; n < projects.size(); n++)
                                        q.addProjList(String)(projects.elementAt(n));
                                q.setNumConditions( (whereconds.size() / 2) );
                                for(int n = 0l n < whereconds.size(); n+=2 )
                                        {
                                        temp1 = (String)whereconds.elementAt(n);
                                        temp2 = (String)whereconds.elementAt(n+1);
                                        q.addCondition(temp1, temp2);
                                }
                        DataStrore.writeQuery( q-, outputStream);

        }
        catch (IOException ioe) {
            System.out.println(ioe);
            ioe.printStackTrace();
        }

    }

}
import javax.microedition.midlet.*;
import javax.microedition.lcdui.*;
import java.io.IOException;
import java.util.*;


public class mm_control extends MIDlet implements Controller
{
            private ExternalDB exdb = null;
            private Display disp = null;
        private Stack screens = null;

    public mm_control()
    {

                screens = new Stack();

                nextScreen( new Introscreen( (Controller) this) );
        }

    public void startApp()
    {
    }


    public void pauseApp() {}

    public void destroyApp(boolean unconditional) {}

        private Display getDisplay()
        {
                if ( disp == null)
                {
                        disp = Display.getDisplay(this);
                }

                return disp;
        }

        public ExternalDB getExternalDB()
        {
                return (ExternalDB) exdb;
        }

        public Displayable currentScreen()
        {
                return getDisplay().getCurrent();
        }

        public void nextScreen(Displayable display)
        {
                Displayable cs = getDisplay().getCurrent();

                if ( cs != null)
                {
                        screens.push( cs );
                }

                getDisplay().setCurrent( display);
        }

        public void lastScreen()
        {
                Displayable display = null;

                if ( screens.empty() )
```

```
                        {
                                display = (Displayable) screens.pop();
                        }
                        else
                        {
                                display = new Introscreen( (Controller) this );
                        }

                        getDisplay().setCurrent( display);
                }

}

import java.util.*;

// this is pretty much a data holding object
// it takes the resulting Datum piece from the query and stores it here so it can be displayed on screen

public class Res

{
   private Vector flds = null;
   private Vector reslts = null;
   private int number_fields = 0;

   public Res()
   {
      flds = new Vector(5,5);
      reslts = new Vector(5,5);
      number_fields = 0;
   }

   public void setNumF(int n) {
                        number_fields = n;
                }

                public int getNumF(int n) {
                        return (number_fields);
                }

                void addField(int n, String s){
                        if(n > number_fields)
                                {
                                // since this is the loop control, this should NEVER happen, and if it does,
something really
                                // strange is going on!
                                system.out.println("Woah! somehow the number of the field being set is > total
# of fields!  see class RES");
                        }
                        else
                                flds.add( (Object)(s) );
                }

                void setField(int n, String s) {
                        if(n < number_fields)
                                flds.setElementAt( ((String) (s)), n);
```

```
            }


            void addValue(int n, String s) {
                        if(n > number_fields)
                                    {
                                    // since this is the loop control, this should NEVER happen, and if it does,
something really
                                    // strange is going on!
                                    system.out.println("Woah! somehow the number of the result being set is > total
# of fields!  see class RES");
                        }
                        else
                                    reslts.add( (Object)(s) );
            }

            void setValue(int n, String s) {
                        if(n < number_fields)
                                    reslts.setElementAt( ((String) (s)), n);
            }



}
public class Resolver

{
   private Vector _dBaseListing = null;
   private Vector _dBaseType = null;
   private int _number_bases = 0;

   public Resolver()
   {
      _dBaseListing = new Vector(5,5);
      _dBaseType = new Vector(5,5);
      _number_bases = 0;
   }

public void setResolverDBs(Vector v1, Vector v2){
            _dBaseListing = v1;
            _dBaseType = v2;
            _number_bases = v2.size();
}

public int getType(String Dname){
            for(int count = 0; count < _number_bases; count++)
                        if( (String)(v1.get(count)).equals(Dname))
                                    return v2.get(count);
            return –1;
} // returns the language of the database indicated
```

Works Cited

1. Goldfarb & Prescod, The XML Handbook Third Edition. Prentice Hall PTR, New Jersey; 2001. Xlix.
2. McLaughlin, Brett, Java and XML. O'Reilly & Associates, California; 2000. 194.
3. Villate, Pitoura, et al, "Extending the Data Services of Mobile Computers by External Data Lockers." Proceedings of the 11th International Workshop on Database and Expert System Applications. IEEE, 2000.
4. Seshadri and Garret, "SQLServer for Windows CE – A Database Engine for Mobile and Embedded Platforms." Proceedings of the 16th International Conference on Data Engineering. IEEE, 2000
5. Ibid.
6. Seydim, Dunham, and Kumar, "Location Dependent Query Processing." Proceedings of the Second ACM International Workshop on Data Engineering fore Wireless and Mobile Access. ACM Press, NY: 2001
7. Goto and Kambayashi, "Dynamic Personalization and Information Integration in Multi-Channel Data Dissemination Environments." Proceedings of the Second ACM International Workshop on Data Engineering fore Wireless and Mobile Access. ACM Press, NY: 2001
8. Ibid.
9. Ozen, Kilic, Altinel, and Dogac, "Highly Personalized Information Delivery to Mobile Clients." Proceedings of the Second ACM International Workshop on Data Engineering fore Wireless and Mobile Access. ACM Press, NY: 2001
10. Ibid.
11. Kuramitsu and Sakamura, "Towards Ubiquitous Database in Mobile Commerce." Proceedings of the Second ACM International Workshop on Data Engineering fore Wireless and Mobile Access. ACM Press, NY: 2001
12. Huang and Garcia-Molina, "Publish/Subscribe in a Mobile Environment." Proceedings of the Second ACM International Workshop on Data Engineering fore Wireless and Mobile Access. ACM Press, NY: 2001

Bibliography

Chase, Nicholas, XML and Java from Scratch. Que Publishing, Indianapolis, Indiana; 2001.

Goldfarb & Prescod, The XML Handbook Third Edition. Prentice Hall PTR, New Jersey; 2001.

Goto and Kambayashi, "Dynamic Personalization and Information Integration in Multi-Channel Data Dissemination Environments." Proceedings of the Second ACM International Workshop on Data Engineering fore Wireless and Mobile Access. ACM Press, NY; 2001

Huang and Garcia-Molina, "Publish/Subscribe in a Mobile Environment." Proceedings of the Second ACM International Workshop on Data Engineering fore Wireless and Mobile Access. ACM Press, NY; 2001

Kuramitsu and Sakamura, "Towards Ubiquitous Database in Mobile Commerce." Proceedings of the Second ACM International Workshop on Data Engineering fore Wireless and Mobile Access. ACM Press, NY; 2001

McLaughlin, Brett, Java and XML. O'Reilly & Associates, California; 2000.

Ozen, Kilic, Altinel, and Dogac, "Highly Personalized Information Delivery to Mobile Clients." Proceedings of the Second ACM International Workshop on Data Engineering fore Wireless and Mobile Access. ACM Press, NY; 2001

Quin, Liam, Open Source XML Database Toolkit: Resources and Techniques for Improved Development. Wiley Computer Publishing, New York; 2000.

St. Laurent and Cerami, Building XML Applications. McGraw-Hill, NY, NY; 1999.

Seshadri and Garret, "SQLServer for Windows CE – A Database Engine for Mobile and Embedded Platforms." Proceedings of the 16th International Conference on Data Engineering. IEEE; 2000

Seydim, Dunham, and Kumar, "Location Dependent Query Processing." Proceedings of the Second ACM International Workshop on Data Engineering fore Wireless and Mobile Access. ACM Press, NY; 2001

Villate, Pitoura, et al, "Extending the Data Services of Mobile Computers by External Data Lockers." Proceedings of the 11th International Workshop on Database and Expert System Applications. IEEE; 2000.