Selective Dissemination of Information in the Dynamic Web Environment

A Thesis

Presented to the Faculty of the School of Engineering and Applied Science

University of Virginia

In Partial Fulfillment of the Requirements for the Degree of

Master of Science (Computer Science)

by

Edward K. O'Neil

May 2001

Approvals

This thesis is submitted in partial fulfillment of the requirements for the degree of Master of Science (Computer Science)

Edward K. O'Neil (Author)

This thesis has been read and approved by the examining Committee:

James C. French (Thesis advisor)

Worthy N. Martin

Ronald D. Williams

Accepted for the School of Engineering and Applied Science:

Richard W. Miksad

(Dean, School of Engineering and Applied Science)

May 2001

Abstract

A selective dissemination of information (SDI) system attempts to facilitate users' information retrieval and information filtering needs. With the rise of the internet as an information source, the volume of information available ranging across all interests has exploded, and difficulties in surveying, querying, and filtering information pertaining to individuals' interests increase with this explosion. The goal of an SDI system is to deliver new information arriving at an SDI-aware information provider to users who express their interests via user profiles. Mechanisms used to implement such SDI systems vary; one such option is a persistent query mechanism. Users create and pose queries to an SDI system; queries remain resident in the system which works to somehow match documents and users. Successful matches are delivered from the SDI system back to users.

In this thesis, we address the dimensions of support necessary to implement a persistent query system; these are considered to be applicable at two locations within such a system - at the information provider and in any system-wide infrastructure that may exist to support persistent querying. For our own protocol, we select a set of these dimensions, compose them, and present our reasons for such decisions. Specifically, we are concerned with handling internet-based information providers, the scalability of the protocol, and not unduly burdening the information providers participating in persistent querying. The solution we propose uses variable rate transmission of change notifications from information providers; we believe these characteristics to be key to the development of a persistent query system. Our protocol is implemented in the existing Personalized Information Environments (PIE) system.

Acknowledgements

1

Introduction

1.1 Overview

Information overload is a pervasive problem in the internet age. An explosion in the variety of information providers including internet web sites, digital libraries, and vast numbers of others motivates a need to facilitate effective information retrieval and dissemination in wide-area environments. The ability of individuals to comprehend and search all of the information sources of interest as a regular task in maintaining currency in satisfying a specific information need is becoming increasingly difficult.

Current technologies for organizing and accessing information are challenged to address this problem and its scale, and fundamental changes in how people view and access information sources will surely evolve in the coming years. One method for addressing this problem is through selective dissemination of information (SDI). A concept that has existed formally since the late 1950's [Luh58], selective dissemination of information is a process by which individuals with an information need express to an SDI system that need in some form; then, individuals receive new information that arrives at information providers participating in the SDI system. User profiles can assume many forms, perhaps a free-text query, a SQL query, or rule sets. SDI is primarily concerned with keeping users up-to-date with information matching a user profile as it arrives at an information provider. The SDI concept has been widely used in the library setting for bibliographic references from tech-

nical literature but has yet to catch on in more mainstream applications, on a large scale, and with a sufficiently fine-grained search mechanism. A more widely used paradigm for disseminating information to users is through information filtering [BC92]; in this model, a continual stream of information from a source is transmitted to all users and filtered by each user's rules to decide which items to actually present to a user.

Clearly, an information retrieval and distribution system supporting SDI could be of great benefit for users. In such a system, a user could arrive at work in the morning to find documents or pointers to documents that relate to a user's profile and that describe information in which the user has expressed interest. This saves the user time and effort and allows the user to concentrate on the pertinent information without having to cull many information providers independently. Selective dissemination of information is suited to providing this solution, and persistent querying is a means by which to implement such a system.

1.2 Persistent Querying

Two models for querying an information provider are possible, and often, an SDI system provides both. First, retrospective querying is the type of query executed by a typical internet search engine[1]. The results of the search are a retrospective look into what the search engine has collected in the past via harvesting information items, such as web pages, indexing them, and adding them to a collection of documents maintained at the engine. The second query model is the one addressed in this thesis, persistent querying. Persistent querying is most concerned with the arrival and dissemination of new information items arriving at an information provider. Unlike retrospective queries which are a one-time query to an information source, persistent queries remain active over time and operate to cull new information from information sources on a user's behalf. This thesis presents a study of persistent querying and a concrete persistent query based solution to the selective dissemination of information problem. A persistent query is created by a human user and

[1]Such as `http://www.google.com`.

expresses an interest that addresses a specific information need. The interest is posed to a fixed or variable set of information providers, and some mechanism matches documents arriving at information providers to expressions of user interest. Persistent queries remain resident in such a system until they expire or are explicitly deleted. The act of informing users of the arrival of new information matching their interests may take many forms such as an e-mail, a page, or a specialized message. The content the user receives may be web pages, scientific datasets, e-mail, news, images, or any other type of transferable content; obviously, copyright laws must be considered in an SDI system. Queries in such a system may be free-text queries expressed by the user, rule based filters, SQL based queries, or machine learned user profiles. A goal of an SDI system is to have the user participate as a passive recipient of information; the SDI system is the active element and should do the work in finding information items and perhaps even finding information providers that match the user's information need. Regardless of how the user expresses interest and how the user receives pertinent documents, the action of expressing the interest and receiving a stream created by matching user profiles to new information items (or vice versa) is a persistent query.

The solution presented here is developed for an information consumer and an information provider where the supplier has a free text search interface and may meet some additional, simple constraints. When the user is logged off from the SDI system, the information consumer, or user proxy, operates on the user's behalf by posing queries to information providers or by analyzing documents. These actors and a scenario are explored in the next section.

1.3 A Typical Scenario

In order to understand how a persistent query operates, it is important to be familiar with the components involved in a simple expression of interest and the distribution of matching information items. The actors involved in a generic persistent query interaction are the

following:

- a human user with a persistent information interest;
- a user's proxy that acts on behalf of the user after the user has expressed an interest in some subject; and
- an information provider that is selected somehow, either by the user or by default, as a candidate for providing information that matches the user's interest.

These three actors interact in an persistent query mechanism in an SDI system to deliver to information items that have been selected for the user based on the user's expression of interest in a subject or according to some user profile. These actors might interact in the following scenario.

The user has an interest in the topic "electoral college," so the user opens up a graphical user interface that provides a point of entry to the SDI system. The user creates a user profile that is persistently stored in a user proxy in the SDI system infrastructure. The user can either select a set of information providers over which to pose the persistent query, though the set of information providers may be fixed for the SDI system in which the query is posed. Once this set has been selected, the user poses the persistent query to the set of information providers, and then the user may close the user interface. The SDI infrastructure is now responsible for somehow matching the user's query to *new* literature that arrives at the information providers. Different SDI systems handle this in different ways and may disseminate the user's profile to distributed information providers, may add the profile to a centralized server, may match user profiles to documents, or may match documents to user profiles. New items accrue at the user's proxy, and the next time the user starts the user interface to the SDI system, the user can view the search results list containing the information items that have matched the user's profile since the user posed the query. Subsequent checks of this list will show all documents that remain undeleted by the user from previous searches and all of the new information items that have arrived since the user last checked the list.

The selection of information providers over which the persistent query may be posed takes two forms. As has often been the case in library SDI systems, the query can be posed over a fixed set of information providers without the option provided to the user to add or remove elements of the information provider set. This treats the information provider set as a black box and has the ability to simplify the user interface because the user need only pose a query, not select the information providers over which to do so. On the other hand, a fixed information provider set may be wasteful if information items of interest exist on only one of ten information providers in the set because the query may be posed to or resident at sources with little chance of receiving matching documents. For instance, a history query posed over a fixed set including a computer science database would likely be wasteful. In the second method, information provider set creation is a user-driven process. In some interface to the persistent query / SDI system, the user is shown a palette of information providers that are available in the system. The user is free to choose all, some, or none of the providers to be members of the set of information providers over which the user will pose a persistent query. Users have the freedom to apply their own knowledge of the providers in selecting the most appropriate way to meet an information need; however, a user may also neglect information providers that may now or at some future time meet the information need simply because the characteristics of such information providers are unknown to the user. Which method of information provider set selection is used will be a decision for each persistent query system, but either way, the system must be flexible enough to allow some administrative authority to add and remove information providers as the provider set changes over time.

The persistent query mechanism is what functions between the time the user poses the query and then reads the search results. Persistent query functionality could simply be a feature of an existing information retrieval and distribution system or may be its own entity, but the characteristics, features, and implementation of such a facility are similar in either case and are what we will consider in this thesis. This is obviously a simple model, and we will build on it as we develop the persistent query protocol further. The above scenario

will evolve as we begin to require features on the information providers in order to support a meaningful and efficient interaction between user proxies and information providers.

There is an interesting item to note when discussing real SDI systems that attempt to provide functionality similar to that just described; systems that provide true information retrieval techniques for indexing documents and query processing are often inversely related to those that scale and handle large user populations. All are desirable characteristics of an SDI system. Information retrieval techniques generally provide a finer-granularity of search capability whereas information disseminated from systems serving large numbers of users is often more coarse-rained. Examples of each will be presented in Chapter 2. One of the goals for the work presented in this thesis is to bring the two together to provide an environment where users have fine-grained control over their searches but have a wide variety of information providers from which to choose. In addition, the type of search interface provided should be left up to individual information providers and the solution should be able to support large numbers of users.

1.4 Thesis

The addition of a persistent query mechanism to an information retrieval and distribution system provides a method for users to receive a stream of documents from an information provider as a provider's contents evolve over time; documents delivered to users will match some query-based expression of a user's information need. The thesis of this document is that a solution to the problem of developing a persistent query protocol exists, and in addition, the solution is lightweight, efficient, scalable, requires minimal support at the information provider, and is applicable to the dynamic web environment. It has been implemented in the PIE [FV99] system.

1.5 Contributions

This work makes several contributions to the information retrieval community.

1: A characterization of the dimensions in which a persistent query may be supported. This is important in describing the levels of support for persistent querying that may be provided by an information provider and those provided by the SDI infrastructure of an information retrieval / distribution federation in support of a persistent query mechanism. The dimensions that can be supported in each location and the tradeoffs for the presence and absence of each dimension are described.

2: A set of requirements for minimally supporting an efficient persistent query protocol. This includes a necessary and sufficient condition for supporting a persistent query protocol without polling.

3: A realization of the protocol in a working system.

1.6 Organization

The remainder of this document is organized as follows. Chapter 2 discusses generalities of SDI and presents past solutions to SDI problems. A continuum of the solution space is defined, and persistent querying is identified in relation to other solutions to the problem; the continuum is presented in terms of the level of content personalization performed by an information provider. Difficulties in doing SDI over the internet are presented. Chapter 3 provides a characterization of the dimensions for supporting persistent querying. Support for persistent querying can exist at two levels, on the information provider and in the SDI infrastructure. This chapter presents different levels of support and guarantees and describes their tradeoffs versus system complexity. The choices for our persistent query mechanism are presented at the end. Chapter 4 explains the optional and required choices of features described in Chapter 3. It abstractly presents the persistent query protocol as it will be implemented and discusses what we consider to be the most reasonable support levels. It explains how the protocol is lightweight, is scalable, and handles dynamic information providers. Chapter 5 introduces the PIE system, the test-bed, and details the steps required to implement the features described in Chapter 4 into a working information

retrieval system. Chapter 6 presents a qualitative analysis and arguments supporting why persistent querying in PIE is scalable, lightweight, and supportive of dynamic information providers. Chapter 7 presents related work considered under the topics of historical SDI, systems work (including active databases and event notification services), and other continuous query work. Chapter 8 wraps up this discussion and presents challenges and opportunities for future work on this and other persistent query systems.

2

The SDI Solution Space and Persistent Querying

2.1 Overview

The solution space for implementing SDI systems includes but is not limited to persistent querying. In order to motivate the choice for persistent querying as a means for solving the SDI problem, it will be useful to explain this space. In addition, this chapter will consider characteristics of internet information providers and the difficulties with performing SDI over such information providers.

2.2 Selective Dissemination of Information

Selective dissemination of information is a concept that originates in the roots of computer science. Luhn originally presented the idea for a system that selectively disseminates information to users based on interests expressed in user profiles [Luh58]. In the years since Luhn described this idea, SDI systems have been implemented in several different ways. Originally, Luhn suggested inferring the user's interests from the information items the system sent the user. In practice, more tractable ways of constructing user profiles have been implemented.

Libraries have been a focal point for implementing SDI systems, which have been used to provide users with updates of bibliographic information from technical journals. Several

different systems of this type had been implemented by the mid-1970 and were used in university libraries; these are summarized by Housman [Hou73]. In these systems, users created profiles that consisted of Boolean phrases denoting the properties recommended documents should have. Often, a high incidence of "NOT" clauses existed in such profiles in order to exclude blocks of documents from consideration. In addition, these library systems allowed users to view and modify their user profiles to fine tune and evolve the search results returned to the user. For efficiency sake, profiles were sometimes grouped based on similar interests of users. In terms of the size of such systems, the discussion mentions systems ranging from fifty to five-hundred profiles and 15,000 documents over 200,000 information items inserted into the SDI system per year. Housman cites one of the significant impediments to the progress of library SDI systems as database compatibility. Interestingly, the expense of SDI systems in research libraries was an issue as a result of shrinking library budgets, which is still a problem today; however, at the time, the bulk of such expenses were devoted to purchasing computer time used to match user profiles to documents. In the thirty years since this survey paper was published, the goals of and problems with SDI systems have not changed, but the number of potential users and potential information providers has exploded in size.

In a traditional sense, SDI systems disseminate documents to users based on the profiles the information disseminators possess about the users. In the internet age, scaling this type of system is infeasible because it requires all information providers to be continually aware of all users' interests, so we evolve the definition of SDI to be the creation of a customized content stream that is delivered to a user based on some type of user profile of which all information providers may not have knowledge. Regardless, the fundamental idea behind SDI is to somehow match *new* documents to expressions of user interest.

2.3 Additional SDI Systems

SDI systems have taken several forms since the mid-1970's. They are in use today in many forms, though perhaps not personalizing content to the degree that Housman [Hou73] describes. In fact, common everyday information streams used by the average internet user can be considered selective dissemination of information mechanisms, although they are usually called information filtering systems [BC92]. These systems form a continuum describing the level of personalization of content that they provide to their users. Here, personalization is defined as the degree to which a user can specify the content of the information stream they would like to receive from the SDI system; a high degree of personalization of an SDI stream means that users can specify at a fine granularity the types of information items they would like to receive. A low degree implies that users receive an information stream that is only broadly of interest to users. At the simplest end of the continuum, the user expresses an interest and information is disseminated to the user with little or no filtering to determine further interests of the user. At the opposite end of the spectrum, machine learning algorithms observe the behavior of users to create user profiles and then scour the internet to discover information items and sources that match the profiles. Systems exist at both ends and at several other points along this continuum.

Providing a low degree of personalization, subscription services send e-mail headlines from some information provider to a subscribed user in a simple form of SDI. Users express interest by subscribing to the news headlines or to a mailing list about a topic of interest, such as cooking or perhaps programming language design. While this expression of interest is very broad, it is still a request for a filtered information stream. The user then becomes the passive entity with the user's e-mail box acting as a proxy for holding search results. At some interval, e-mails arrive containing content the user has requested, and the user can filter the information items of interest from the rest of the stream. This is probably the simplest SDI system today, and its mainstream use is common. These types of subscriptions have a low degree of personalization of the content stream.

In the next step in terms of the user's ability to personalize an SDI information stream, systems such as the Pointcast [RD98] network use a user interface and push technology[1] to implement SDI. Users select the content they wish to have pushed to the proprietary Pointcast client executing on a user's desktop. While little information persisted for the user in the system, users were continually receiving a filtered stream of new content rendered to their desktop during the execution of the Pointcast client. Users select categories in which to receive information such as current news headlines, sports scores, and technology or business news. More recently, this type of SDI has been moved into the web browser and out of a proprietary interface. Internet portal sites such as Excite[2] and Yahoo![3] provide a similar degree of personalization of the SDI stream. The web page displayed in a user's browser is periodically updated to reflect the newest content available from the content provider. Such portal sites act as a focal point for additional information provider's content streams; Excite for instance provides content from the AP, Dow Jones, Reuters, and other news services integrated based on content for each subtopic, news, business, sports, and so on. Thus, users receive content from many different information providers, which has been pre-categorized and packaged. The degree of personalization of this content stream is higher than an e-mail broadcast, and the user filters the stream by clicking on the items of interest appearing in the browser. Still, this stream is topically specified but at a coarse granularity.

A more recent advance from traditional portal sites are features such as those provided at Octopus.com[4]. This web site provides a service whereby users can create their own portal pages starting not with pre-created categorizations of topics but by selecting their own information providers and then integrating them into a customized page. This is an even finer granularity of control for personalizing the content stream; by selecting the information providers and specifying the layout, users are in effect posing queries over those

[1] Franklin and Zdonik [FZ97] have noted that the use of push technology in Pointcast is actually very limited and that the content is delivered through a pull mechanism initiated by the Pointcast client.

[2] `http://my.excite.com`

[3] `http://my.yahoo.com`

[4] `http://www.octopus.com`

content streams and are just not selecting the individual, broad topics from the streams.

It is at this point that the inverse relation, mentioned in Chapter 1, between scalability of an information distribution system and the information retrieval abilities of the system tradeoff. Previous mechanisms support millions of users of news groups, mailing lists, and portal sites with millions of daily information items; however, systems with more sound and capable information retrieval based search techniques are often presented in terms of thousands of users and information items. The library scenario described above and presented in Housman [Hou73] and other literature of the time is a large step along the continuum of personalization when compared to portal sites. In such systems, users could further personalize the stream of literature they receive by creating a profile and specifically tuning the stream to match an information need. Pasadena [WF89, FW91] operated over Netnews and delivered indexed articles to users over a wide area network; Pasadena later added SDI functionality [FW91]. More recently, the Stanford Information Filtering Toolkit (SIFT) [YGM95] is an effort to create an efficient and scalable solution to the information filtering / SDI problem; SIFT also operates over Netnews articles. Users interactively create a profile and test it against test collections using retrospective queries. When the profile satisfies the user, the user e-mails the profile to a central server that evaluates the profile at a time interval specified by a user. The profile is evaluated periodically against all accumulated Netnews messages, and articles matching a user profile are sent to the user's e-mail address. This system uses information retrieval indexing techniques from the WAIS [KM91] toolkit to allow users, as in the library setting, to post user profiles and receive periodic results. This is an advance over previous items in the continuum because information retrieval techniques are used to compare a user query to documents. This clearly can provide a higher fidelity information stream than is available at a portal site or via a newsgroup subscription.

The final step along the continuum of SDI solutions are those created in the early- and mid-1990's using agent technologies [SM93, Mae94, Lie95, BS95, BSY96, Bal98] ; such systems provide the possibility of a high degree of personalization. Researchers developed

software agents to observe a user, infer the user's interests, and then discover and filter information providers such as web sites [Lie95, BS95, BSY96, Bal98], e-mail [SM93, Mae94], or internet news [Mae94] to provide users personalized streams of information. Such systems often employed technology more advanced than a user's development of a static query posed over the incoming information stream. Machine learning techniques including neural networks and genetic algorithms are used to observe, statistically quantify, and evolve a user's interest in topics the agent systems noticed that users frequented. User profiles created in such a manner are more adaptable to changes in user interest because they can evolve over time as a user's interest evolves and because they do not rely on the user to update their query. Their dynamic nature absolves the user from explicitly maintaining them as is the case in many of the other steps along the continuum, a problem noted by Housman [Hou73]. The success of such systems is mixed because of difficulty in training and evolving user profiles; however, it is still an interesting idea and was actually the one that Luhn [Luh58] originally described for profile generation. Such systems have never been scaled to support a significant user base. The degree of personalization provided by these systems has the *potential* to be excellent.

Our view of persistent querying is most similar to the library view of SDI, but because of how we will realize persistent querying, we provide users even more ability to personalize the stream of documents delivered from the SDI system to the user because of the flexibility in supporting wide varieties of information providers. Our solution exists in between the information retrieval and agent-based systems along the personalization continuum.

2.4 Internet Information Providers

One point to note in this discussion is that the traditional idea of selectively disseminating information from a few information providers is a very different type of problem from selectively disseminating information from tens of thousands (or more) information providers to millions of users; the characteristics of the information providers are often very different.

We consider an internet information provider to provide content at "internet time" in an internet accessible format. This simply means that the rate of content change at such a provider is larger than it has been in the past for such providers as library databases. In addition, internet information providers often have the unfortunate characteristic of being lossy. This means that the provider will have a given document in its database at one time and have lost it at a "short" time later. Quantifying this rate of change and the degree of loss is not our concern here and is nigh impossible, but it should be easy to accept that many information providers on the internet are in a constant state of flux in terms of their gain or loss of information items. In addition, many of the providers that we are interested in may be uncooperative in that they do not provide support for a persistent query system. The goal of our persistent query protocol is to capture as much of this content as quickly as the user deems necessary to resolve the risk of its loss. Guaranteeing unconditionally that a user will see every relevant document that ever arrives at an information provider of interest will be virtually impossible, and it should be easy to see without significant argument that this is the case, especially at providers that do not aid in supporting a persistent query system. Meeting this requirement in a library environment is easier because citation indexing and dissemination SDI services do not have to deal with the degree of loss incurred with internet information providers. Our persistent query protocol attempts to to resolve the risk associated with document loss in the greatest degree possible without unduly burdening providers in a persistent query system. Mention of an internet information provider should imply that such a provider is not necessarily providing significant support for persistent querying and is not guaranteeing to preserve information items for an indefinite amount of time. We refer to "well-behaved" internet information providers which are those that when presented with a document store the document for future retrieval for a "reasonable" amount of time on the order of days instead of minutes. Realizing that the argument and definition are qualitative instead of quantitative, it should be plausible enough to accept as a reasonable description of a general internet information provider.

2.5 Persistent Querying

Having discussed the solution space for selectively disseminating information and defined the types of information providers of interest, the foundation has been laid for describing the requirements of a persistent query system. A persistent query system that exists in such an information provider environment must be lightweight, easy to implement, flexible, and tailored to meet users' information needs. A persistent query protocol must be lightweight in terms of the overhead required at information providers and at the user proxy operating on behalf of a user in the SDI system. In addition, because of the scale on which we are considering this system, the communication frequency between user proxies and the information provider must be kept to a minimum so as not to unduly burden the network and the computational resources at each. Second, the system must be easy to implement or integrate into an existing information retrieval system or information provider. This is especially true of the information providers participating in a persistent query system. Burdening providers unduly with requirements must be avoided in order to decrease the cost of participation for information providers. In order to implement persistent querying, however, simple requirements will be placed on information providers; these will be defined and discussed in Chapter 3 with our choices for a persistent query protocol presented Chapter 4. Failure to support these requirements will not prevent an information provider from participating in the SDI system, but it will remove it from the scope of the information providers we are currently considering. Third, the system must be flexible enough to adapt to the changing needs of users and to adapt to changes in the set of information providers participating in the persistent query system. Because we expect (in the worst case) the information providers to be operating on the internet, the persistent query protocol must account for this and be as adaptable as the internet is dynamic in terms of accepting new information providers without overhauling persistent query system features to do so. Finally, the system must allow users to express information needs in such a way that those needs can be met by the information providers. From our perspective, this is possible by

allowing users to express free-text queries to user selected sets of information providers. The assumption is that users are better at expressing and meeting their information needs than fixed content streams or the current machine learning techniques for inferring a user profile, and providing users the ability to free-text query an information retrieval search engine is an effective avenue for serving information needs.

2.6 Conclusion

A survey of the solution space for SDI systems has been described, and the location of a persistent query system in the continuum has been noted. Because of the proliferation of information providers on the internet, this information rich environment must be considered when constructing a persistent query system. As a result, the unique characteristics of internet-based content providers should be considered when designing a persistent query system. A persistent query protocol implemented as a solution to the SDI problem should be able to handle internet information providers and must be lightweight, non-intrusive, flexible, and able to allow users to express and meet their information demands. The following chapter will present dimensions that can be implemented to support persistent querying and are split between two locations, those at the information provider participating in the persistent query system and those in an infrastructure supporting persistent querying (if such an infrastructure exists). Later chapters detail the design of such a persistent query protocol and describe its implementation in a working information retrieval system.

3

Design Dimensions for Persistent Querying

3.1 Overview

SDI systems match new information items to expressions of user interest. Fundamental to such systems is the task of detecting the change of content at information providers because this is a precondition to discovering and delivering new content to users. SDI systems typically consist of two major elements, information providers and user proxies. Information item distribution from provider to user proxy is based on some kind of user interest profile. The profile describes the types of information in which a user is interested; free-text queries are of particular to us. Unlike some previous solutions to the SDI problem [GNOT92, YGM95, PL98], free-text queries can tailor the information stream to users' interests using information retrieval techniques for querying as opposed to database query methods. Given the pervasive nature of publishing on the internet, a persistent querying SDI system should operate both on a small scale and on an internet scale.

An issue in supporting persistent querying is formulating a set of requirements to place on information providers wishing to participate in the persistent query mechanism. Current and previous systems have focused on providing persistent query support for a single information provider or small provider set. A difficulty with this approach is that the framework for supporting change detection may be closed or may be specific to each information provider type in the SDI system, of which there are often few. Such systems do not scale

well and are difficult to evolve and supplement with new information provider types and instances. An infrastructure called a federation can be provided to address these latter two problems. By placing a set of standard and open requirements on information providers and implementing such requirements in a scalable manner, a persistent query mechanism can facilitate a wide range of participants in an environment that can be widely utilized. This chapter discusses the levels of support that can be implemented by an information provider participating in persistent querying and the implications of each dimension. Dimensions of support are grouped based on where they can be located, at the information provider or within a larger federation infrastructure supporting persistent querying. Identifying these levels of support is a research goal of this thesis.

The perspective we take here is predominantly theoretical and simply presents major dimensions along which persistent queries might be supported, how this functionality might operate, and what is possible in such a system. One specific goal must be kept in mind: requirements placed on the information providers participating in persistent query operations must be kept to a minimum. On an internet scale, providers will be required to service tens of thousands to millions of users and their proxies, so scalability, especially in terms of processing and storage requirements, must be considered when discussing the support that an information provider can implement. Persistent queries can be supported along six dimensions:

- Information provider-based change notification emission;
- Timestamping items inserted into an information provider's content;
- Provision of a search interface at an information provider;
- Caching of information items;
- Supporting guaranteed change notifications; and
- Supporting guaranteed immediate change notifications.

These dimensions can be divided into two groups; the first consists of the initial three elements in the above list, which the information provider may provide natively. The last three are levels of support that require additional infrastructure outside of the information provider[1]. In this last set, a federation infrastructure supporting the dimensions is necessary. Justification for these claims will be provided in the following sections. Note that the first three items can also be provided, at a certain expense, by the federation infrastructure for uncooperative information providers that do not implement them natively. Several terms have already been used that require definition such as information provider, notification, and guarantee; these and the notation used to represent sets and instances of important elements follow.

3.2 Definition of Terms

Information provider - An information provider is a source of information items that is administered via some authority. Minimally, information providers have a browsable interface facilitating access to their information items. Information providers are "well-behaved," a term explained in Section 2.4. Information providers are not proactive in a persistent query system and simply provide services that can be used by active elements of an SDI system. In particular, it is not assumed that information providers provide an SDI capability.

User proxy - A user proxy is an entity that facilitates a user's persistent querying of information providers. The user proxy is generally persistent and stores a persistent queries and search results on the user's behalf. The user proxy is the destination for messages sent from information providers that describe the state of the provider's content. A user proxy is an active element in a persistent query system and operates on the user's behalf when the user is not actively interacting with an SDI system. User

[1]Actually, the information provider could support the latter three dimensions in lieu of the use of federation infrastructure, but requiring any one of these dimensions at the information provider we feel unduly burdens the provider.

proxies have a unique, SDI system wide identifier that is used by other participants in the SDI system to communicate with the user proxy.

Persistent Query - A persistent query is a user's expression of interest in a topic that is realized in a query syntax which can be structured or unstructured. The persistent query is posed over a fixed or variable set of information providers; as content changes at information providers, the persistent query model dictates that users' persistent queries are evaluated at the providers, and the search results are returned to user proxies. A basic persistent query consists of two parts, a unique query identifier (unique relative to a single user's proxy) and a free-text query. We assume free-text queries throughout this thesis. In our discussion, the query is fixed during its lifetime at an information provider.

Change notification - A change notification is a simple message that denotes a content change in the documents stored at a single information provider. Such a notification is sent from an information provider to a "subscriber," an entity that has registered an interest in receiving a change notification. In our discussion, all subscribers to a change notification service of an information provider will be user proxies. A change notification does not contain search results for a user's persistent query, and the presence of a notification does not imply that documents useful to a user's query have been added to the content at an information provider. In addition, notifications do not contain or imply sequence numbers. Change notifications only signal a content addition[2] at an information provider.

Timestamp - A timestamp is a marking that can be assigned by an information provider to information items inserted into the provider's content set. Timestamp values must be monotonically increasing values so that no two items ever have the same timestamp. A timestamp provided to a document, d_i, is t_i, where $d_i \in \mathcal{D} = \{d_1, d_2, ..., d_{|\mathcal{D}|}\}$

[2]Note that deletions are also a form of content change at an information provider, but our model does not consider change notifications when deletions occur because SDI focuses on *new* content.

and $\mathcal{D}$ is the set of documents at the provider at any given time. The following relationship must hold, $d_i, d_j \; s.t \; i \neq j \wedge t_i \neq t_j$; timestamps and documents exist in a $1:1$ relationship. In addition, timestamps are not meant to convey a global notion of time and are only used to provide a total ordering of documents at a single information provider.

Search interface - A search interface is a means by which some client can pose a query that is evaluated over the information items indexed at an information provider. The result of a query is a search result containing a ranked list of information items chosen by the information retrieval technology powering a provider's search interface. In this thesis, we are not concerned with the effectiveness of the results of such a search; the search interface at an information provider is treated as a black box which operates as described. A search interface is stateless, especially in terms of persistent queries, user profiles, and the results of user queries.

Federation - A federation is a system-wide infrastructure that is deployed to support persistent querying and other information retrieval and distribution operations. The persistent query components defined so far (information providers, user proxies, etc.) comprise a peer-to-peer system between providers and user proxies. In order to facilitate functionality in addition to that provided by independent information providers, a federation is necessary to bring information providers together within a common framework that provides support and that can provide value-added services to users and their proxies in persistent querying. This infrastructure may build guarantees on top of the persistent query support dimensions available at an information provider. The participants in a federation vary depending on the federation's requirements, but all federations minimally consist of information providers and user proxies.

Information Provider Proxy - An information provider proxy is a wrapper for an information provider and is used to export an information provider's functionality to participants in a federation. For example, if change notifications are implemented

at an information provider, the provider's proxy assumes the responsibility for handling subscriptions and cancellations for other entities in the federation. Information provider proxies exist in a $1:1$ relationship with information providers participating in a persistent query system. A provider's proxy does not exist in a persistent query system without a federation.

Cache - Because of the dynamic nature of web information sources, a cache may be useful for saving documents of interest to the user if a user is unavailable to view them. Caches can exist at many places throughout an SDI system including at the information provider, a provider proxy, or a user proxy. The use of a cache is generally a policy level decision that will be left up to some administrative authority.

Guaranteed - One design option for a persistent query system is to operate by transmission of change notifications from information providers to user proxies. The simplest notification delivery system implements a peer-to-peer protocol that does not require the infrastructure of a federation and operates in a datagram style, unreliable delivery mode. This design, however, does not provide strong guarantees assuring the delivery and timeliness of a change notification. In a simple guaranteed model, a notification is assured to eventually arrive at a user proxy under certain, reasonable operating conditions. We assume that a federation infrastructure is necessary for providing such a guarantee because placing the requirements for assuring the guarantee on an information provider is an unreasonable burden. An information provider's proxy is used in an infrastructure to facilitate delivery of guaranteed change notifications and increase scalability of the SDI system. Conditions on the guarantee are discussed in Section 3.4.2.

Immediate - In addition to providing guaranteed notifications, notifications may be delivered "immediately" to each element of an information provider proxy's subscriber set. Immediate delivery does not imply that delivery is real time, but it does guarantee with certain conditions that a notification will be delivered to a user's proxy

without significant waiting. Immediate mode and guaranteed mode are analogous to the postal service leaving a letter in a mailbox and delivering the letter by hand, respectively. Again, an infrastructure is assumed in order to support immediate notifications without unduly burdening an information provider. Conditions on the immediacy aspect of the guarantee are discussed in Section 3.4.3.

The subsequent discussion will be decomposed into two parts. First, those persistent query support dimensions which can reasonably be made on an information provider are described; these include change notifications, timestamps, and a search interface. Then, the dimensions that require the additional support of a federation infrastructure are described; these include caching, guaranteed change notifications, and immediate guaranteed change notifications. For each dimension, the discussion starts with a definition, describes the implications of the absence of such a feature, and then expands on the implications of the presence of the dimension. In cases where appropriate, a set of method signatures are provided to programmatically show how a dimension might be provided in a persistent query system.

3.3 Information Provider Support Dimensions

Information provider support dimensions are those that may be implemented at the provider without any support from additional entities or a federation. The model under consideration for simple persistent query support is shown in Figure 3.1.

In Figure 3.1, there are two main participants, the information provider and the user proxy. In the most generic case, the information provider has a means by which to browse the content that it stores. The user proxy, through some persistent query mechanism, receives change notifications or a stream of information that has been evaluated against a user's profile. In supporting such interactions between user proxies and information providers, the providers may, independently of each other, have functionality in one of the following three categories:

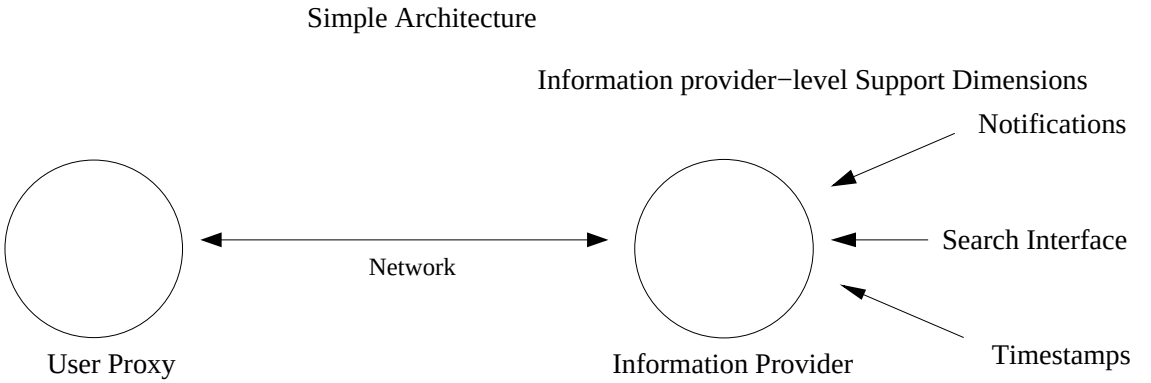

Figure 3.1: Dimensions for supporting persistent queries at information providers.

- emission of change notifications;
- timestamping (or sequence numbering) information items added to their content stores; and
- provision of a search interface.

We will discuss each item in turn.

3.3.1 Change Notifications

Because the SDI and persistent query systems are most interested in new content arriving at an information provider, knowledge of the event of new content arrival is of key importance. There are two general solutions to the problem of conveying this event to interested user proxies. First, the provider may store the user queries and push to the appropriate user proxies new information items arriving in its content store that match the queries. Second, the information provider may transmit simple messages denoting content change to entities that have "subscribed" to receive such messages. We do not consider the first class of solutions because we believe that such a configuration overburdens information providers and that this model will not generally work on internet scale. A subscription based change

notification service is the dimension we consider. Information providers propagate change notifications to the provider's subscribers if new information item(s) are added to the provider's collection. Notifications are sent to each element of the provider's subscriber set[3]. An item in the set contains identifiers used to resolve to a single user proxy and a unique query at that user proxy. Let this set of subscribers be denoted by $\mathcal{L}$ where $\mathcal{L}$ consists of items l_i that have the following attributes:

$$l_i = \begin{cases} \textit{subscriber\_id} & : \text{ subscriber identifier} \\ \textit{query\_id} & : \text{ persistent query identifier} \end{cases}$$

An information provider may choose to notify subscribers of its changes by propagating a message containing the address of the provider (the origin of the notification) and the query identifier of the notified persistent query. If a single subscriber has n queries registered at a provider, then the provider's $\mathcal{L}$ set has n distinct elements for that subscriber with the same *subscriber_id* entry but each with a different *query_id* entry. At the subscriber, the arrival of this message is enough to denote a change, as the provider is assumed not to have other interactions with the subscriber. At the time of a provider's content change, a change notification will be propagated to all elements l, $l \in \mathcal{L}$. Upon receipt of a change notification, the expected behavior of a user proxy is that the user proxy can choose to react to the notification immediately or at a later time; when the user proxy does react, it will send the user's query to the notifying information provider and receive in return a search result containing the *new* information items that have arrived at the information provider.

In the absence of notifications, the alternative is to implement functionality where a user proxy will poll an information provider of interest to try to detect changes to the provider's content. These changes may be detected as differences in word count, byte count, document count, or other characteristics that can be discerned by some means; some of these methods

[3] A "subscriber" may be any entity that subscribes to receive change notifications, but the "subscriber" definition in this thesis implies only user proxy objects as the recipients of change notifications. This specific case generalizes easily so any other participants may receive notifications.

[LPT00] have been used before. The disadvantage of polling is that it generates a great deal of network traffic that may turn out to be useless because of the number of polls that return having detected no changes. Polling is also a burden on the computational resources at the server in having to respond to the polls and at the many clients doing the polling. Previous persistent query solutions [LPT99a, LPT99b, LPT00] have used polling to discover changes at information providers, but clearly, this solution is not the most efficient.

Assuming that information providers send change notifications to subscribers, the information provider can use several different policies to determine the rate of notification emission to the subscriber set, $\mathcal{L}$. The delivery of a notification is assumed to be unreliable, meaning that a notification(s) may not be successfully delivered to a subscriber; such failures will occur independently of each other. This assumption is necessary in order to avoid burdening information providers with error recovery requirements and to reduce a provider's cost of entry to participate in a persistent query system. A notification can successfully be delivered if two conditions exist:

- a path exists through the network from subscriber to information provider (and vice versa); and
- both the subscriber and information provider are alive on the network.

Notifications can be transmitted at a rate defined by one of three models:

1. notify-per-change, which emits a notification in a $1:1$ relationship with a provider's content changes;
2. counter based notification, which emits a notification every n document arrivals;
3. timer based notification, which emits a notification once at the end of the interval t; or
4. variable rate notification, which emits notifications based on the arrival of documents and the response time of the subscribers.

Under notify-per-change, every arrival of a new document at an information provider causes that provider to emit a notification to every element in its subscriber list, but a high document arrival rate incurs a high frequency of notification propagation which may overwhelm the computational and network resources of the system. This method is effective, but it is also overkill because for the average query, every document arriving at a provider will not be useful to every query. In addition, the user proxy receiving the notification will be forced to respond somehow to every incoming message even if the user proxy does not choose to react immediately and pose the query at the information provider for every change. The information about each registered query necessary to support this notification rate is simply that listed for l_i.

The second method, counter based notification, emits a notification to an l_i item after n content changes, where n is a value that can be specified by each subscriber. This allows a user proxy to coarsely control the rate at which an information provider transmits change notifications for a given persistent query. It reduces the amount of network traffic by a factor determined by the sum of the decrease in frequency of notifications for every $< subscriber\_id, query\_id >$ pair. The problem with this model is that each subscriber may wait indefinitely for the n^{th} notification to arrive from the provider. For example, if n is set to five and four new documents have arrived but the fifth will not for some time, the subscriber waiting for the n^{th} notification will wait this full time duration. This notification model also requires the information in the l_i for each subscriber, and additionally, it requires a counter for every l_i element that is incremented each time a document arrives at the provider. The counter is reset when the n^{th} change occurs and a notification is finally sent to the $< subscriber\_id, query\_id >$ pair. An additional problem with this model is that failure to deliver the n^{th} notification (because of unreliable communication) may greatly affect the ability of a subscriber to keep up with information provider changes.

The third method is the timer based notification model and provides an interface by which the subscriber may set the time interval, t, over which it would like notifications to accumulate before being notified by the information provider about a change. For each

subscriber, once t has passed and if any new documents have arrived, the information provider propagates a change notification. As with the counter based method, the issue here is that a document may arrive immediately after the previous timer expired, and the notification for the document will have to wait for virtually the entire interval t to pass before the notification can be sent to the subscriber. In addition, the complexity of the data maintained for each subscriber increases to include a timer per registered query. In a large system, maintenance of this number of timers will become expensive in space and runtime complexity, and recovery from runtime failures of the timer mechanism on the information provider will be expensive or impossible. For each subscriber in the $\mathcal{L}$ set, this method requires the properties maintained in the l_i structure and a running timer and interrupt handler.

The final notification model sends as few change notifications as possible and relates the transmission rate to the rate at which subscribers respond to notifications. We term this the variable rate notification model. In this case, notification emission rates vary based on the arrival of documents at the information provider and the rate at which a user's proxy chooses to respond to them, which may differ for every subscriber. In this situation, the information provider need only to store the data referred to as l_i and one bit called the "notification pending" flag. The bit denotes whether or not the user proxy has been sent a notification since the last time it responded to one. When a user proxy arrives to register a query at the information provider, the bit is set to zero. When the information provider receives a new document, a change notification is propagated to the newly subscribed user proxy. Because no notifications have been propagated to the new subscriber, its "notification pending" flag is false, so the notification is sent to the subscriber and the "notification pending" flag is set to true. Now, the onus is on the user proxy to respond to the notification whenever the user proxy deems it necessary. Until the subscriber handles the last notification it received, additional notifications from the same information provider yield no additional information in terms of change at the information provider and are not transmitted. Each time a notification is to be propagated by an information provider, a

check is made of the "notification pending" bit for each $< subscriber\_id, query\_id >$ pair before transmitting the notification; the transmission is made only if the flag is false for a given pair. When the user proxy finally responds to the change notification it received, all of the documents added since its last response time are available through the query or browse interface of the information provider regardless of how many changes have occurred. The "notification pending" flag is turned to false at query completion time. In the event of a change notification delivery failure, the provider will not receive an acknowledgement packet from the subscriber and will not set the "notification pending" bit to false. Thus, subsequent notifications will propagate to a subscriber until it successfully receives one, but in this simple model, no attempt is made on the part of the provider to re-deliver after a failure. The subscriber still has the option of querying the information provider at any time, which places the burden of recovering from failures on the subscriber rather than the information provider. This makes failure recovery a policy issue; one such policy is to have the subscriber always query the information provider upon recovery. That way, any time that the subscriber is capable of listening for change notifications from information providers, it receives the most current notifications.

The variable rate notification model is a clean model for notification delivery, and it delivers a minimal number of notifications relative to the maximum of one per document per $< subscriber\_id, query\_id >$ pair and has excellent error recovery properties. In addition, it requires simply one bit more of storage per subscriber on the provider than the minimal amount of storage necessary in the notify-per-change model and also requires no running processes for each l_i. Providing variable rate notifications is not a good design if documents arriving at the information provider have short lifetimes because during the interval between user proxy responses, documents may disappear; however, because the responsibility for responding to notifications rests with the user proxies, the user proxies must take steps to resolve these risks to levels that are acceptable to the subscribing user proxies. The opportunity to respond to every change notification immediately is not precluded by this method, but for those user proxies who do not wish to incur the overhead of doing so, the

benefit is reduced network traffic and not having to respond to every document insertion at the provider. Furthermore, we assume that information providers are "well-behaved" in terms of document loss.

Figure 3.2 shows the interface for a mechanism facilitating user proxy subscriptions and change notification delivery.

```
public interface ChangeEventManager {

  public void addChangeListener( ChangeListener listener );
  public void removeChangeListener( ChangeListener listener );
  protected void fireChangeEvent();

}
```

Figure 3.2: Subscription interface at an information provider providing notifications.

The first two method signatures provide the ability for an interested subscriber to register and unregister an interest in change notifications. This interface is suitable for the first and last notification models described above. The subscribe method may require additional parameters; for example, the timed and counter based notification models require interval and count information, respectively. The same functionality in the method signatures above may also be provided through HTTP / CGI or other non-programmatic interfaces.

3.3.2 A Search Interface

Another feature that may be implemented at the information provider level is a search interface. A search interface facilitates access to some or all of the content maintained at the site in an indexed and queryable representation. The provision of a search interface relieves the user proxy from having to manually browse an information provider each time the user proxy polls or reacts to a change notification from the provider. The search interface of the most interest is that in place at a typical internet search engine, for example Google[4], which facilitates free text queries (perhaps using some Boolean syntax) from an inquiring

[4] `http://www.google.com`

client[5].

A search interface is an important feature for an information provider to implement. In addition to allowing a user to retrospectively query a site for documents relating to a given topic, a search engine organizes and increases the usability of the often vast amount of content that is available at a typical internet web site. Search tools are continually evolving and incorporating advances made in the information retrieval field. As search engines evolve and improve, the usefulness of documents returned to users can evolve independently of any SDI mechanism that is employed. The search interface at an information provider need not have any more functionality than free text searching. It is assumed that the query syntax used to initially pose a persistent query will be passed to the search interface to produce a search result consisting of a list of documents ranked highly by the search engine relative to the query. The performance of the search engine in terms of returning relevant results is not of interest in this discussion, we are only interested in the absence, presence, and implications of a search interface. A search interface operates as follows. An information provider has an indexed document collection $\mathcal{D}$ and all documents, $d \in \mathcal{D}$, are accessible via the search interface. Given a query, Q, Q is evaluated at the search engine. The search interface returns $\mathcal{D}'$ where $\mathcal{D}' \subseteq \mathcal{D}$ and $d' \in \mathcal{D}'$ was ranked highly for the query Q through the provider's search engine. This search result, $\mathcal{D}'$, is returned to the user proxy and can be processed additionally there as required.

Absence of a search interface makes persistent querying of an information provider more difficult. A search interface is convenient because the provider is able to return documents spread throughout the site as useful to a query. Without this ability, user proxies interested in a provider's content must shoulder the burden of having to scour the provider for relevant changes that have occurred. In addition, the user proxy becomes burdened with having to determine the usefulness of information items found at the information provider, which will be more difficult than using an information provider's native search interface. Later, in the discussion on federation infrastructure, we will see how a search interface can be provided

[5]The free-text / Boolean query syntax if preferable over SQL or another more structured syntax.

at the federation level. Regardless of how search capability is facilitated for the information provider, it will be very expensive and less effective than if the provider implements this functionality natively.

Assuming a search interface exists, the issue of who stores and initiates execution of a query is important. Query processing will always take place on the information provider, but it can be initiated by either the information provider or the user proxy. In the former case, the provider has the option of sending documents to the user proxy only when an information item ranks highly in the search result produced by evaluating the query at the provider's search engine. This design could be a useful feature because it saves the user proxy from having to come and repose the query to the provider each time a document arrives, and only those documents deemed of interest to the user proxy are returned. Instead of executing each query against the provider's content for each new document, the provider may optimize the task and query the profiles with each new document to match documents to queries. This saves having to run the search engine over the entire contents of the collection for every new document arrival, which could be an expensive operation. Also, there is no need for a provider to emit change notifications if all user proxies explicitly register a query predicate instead of simply a query identifier at the provider. The difficulty with this solution and the optimization, however, is that the information retrieval properties of the search engine may work against good evaluation of the usefulness of a document to a query because the document can not be ranked relative to others like it. The user proxy is best left to treat search results as recommendations from information providers to further evaluate the usefulness of a document relative to a user's information needs. In addition to issues with the information provider solely determining the usefulness of documents, storing the queries on the information provider places burdens on its infrastructure. Each provider must keep track of the l_i structures and the full user query. This is necessary because the provider must be able to process the query and to resolve the user proxy and query pair to which to send the search result, $\mathcal{D}'$. Also, the provider is left to make the policy decision about when to execute the queries against new content. User proxies lose

the ability to execute the query based on their own policies and are left at the will of the provider's administrative authority to enforce a query execution policy. Several systems [FW91, GNOT92, YGM95, LPT99a, LPT99b, LPT99a, LPT00] implement such persistent query functionality.

One result of this is that the information provider is ill suited to evaluate the final usefulness of the document to a user's query and broader information need. This is the case even if leaving this final judgment to the provider is computationally more efficient. Instead, the task of further identifying and acting upon the usefulness of a document is best left to the user proxy, which can leverage its local computational resources, understanding of the user's information need, and information retrieval functionality to determine the final usefulness of an information item to a user's information need. For example, the user proxy may have a more advanced user profile against which to filter documents or may have some comparator and decision process based on the user's document viewing history. Finally, the user proxy initiating the query process guarantees that the user proxy is ready to receive the results, which it might not be if the information provider simply pushes search results asynchronously, perhaps when the user proxy is not active in the interaction. A search interface is useful for retrieving documents out of the whole content at a provider, which results in a rough estimate of the information item's subject matter and results in a ranked list of information items. Additional processing of information items for content is best left to the user proxy.

The other location for the query is to leave it at the user's proxy, which requires the user's proxy to explicitly return to the information provider to issue the query at some interval. This removes the issues just noted of having the providers perform query executions, and it returns control of the frequency of response to the information provider to the user proxy's policies. The queries also do not have to be stored at the provider, and the leisurely nature of notification handling by some user proxies may reduce the load on the provider. Most importantly, the user proxy gains the ability to exploit the search interface of the provider to its fullest potential because it can do information retrieval processing in

addition to that performed on the provider using full result sets instead of only the items the information provider deems useful to the user queries posed through a black box search interface.

A provider's search interface could be accessed in several ways, through HTTP as at an internet search engine, or programmatically using an interface similar to that in Figure 3.3

```
public interface SearchInterface {

  Document[ ] giveAllDocuments( Query q );

}
```

Figure 3.3: A simple search interface on an information provider.

3.3.3 Timestamps

Providing timestamps or sequence numbers to individual information items at an information provider can be used to optimize query processing at the provider. Timestamps must provide a total ordering of all of the content available at that location and must be monotonically increasing to ensure that no two documents will ever have the same timestamp in a content base. A timestamp[6] says nothing about the persistence of a document at the provider.

A timestamp for a document, d, is t, where $d \in \mathcal{D}$ and $\mathcal{D}$ is the set of documents at the provider at any given time. The presence of timestamps enables "differential querying." Differential querying is an algorithm where each time a query is issued over a provider's evolving content, the content is partitioned into two sets based on the query's last execution time, and the query executes only over the partition containing content that has arrived since that time. Differential querying is made significantly easier when timestamps are coupled with an information provider's native search interface, and this discussion will

[6] Timestamps do not have to be based on clock time; any monotonically increasing naming or numbering scheme will suffice.

assume that this is the case. Timestamps could be used when browsing documents because the contents of the information provider may be viewed in order of their timestamps; however, timestamps interact with a search interface in a more meaningful way. Use of timestamps in searching requires the timestamp of the last query execution time to be stored somewhere in the persistent query mechanism, either at the information provider or at the user proxy.

The presence of timestamps when coupled with a way to incorporate them into a search or browse operation is a significant step toward supporting persistent queries efficiently and effectively. The provider's collection can be partitioned on the last execution time of a query, as previously discussed. Because timestamps are distinct and the comparator for placing a document in a partition is exact, no document could successfully be considered for membership in each partition simultaneously. Differential querying is made possible in this situation. Through differential querying and with "well-behaved" information providers, all of the documents present at an information provider that match a registered persistent query will be found by the query at the time of its execution over the partition containing new content. Timestamps and querying interact as follows. A document, d, arrives at a provider that uses timestamps. It is assigned a timestamp t and inserted into the set of documents $\mathcal{D}$ at location i, where $i = |\mathcal{D}| + 1$. Under the definition of a timestamp, for all t ever assigned to any member of $\mathcal{D}$, $t_i \neq t_j, \forall i \neq j$. All documents are now totally ordered in $\mathcal{D}$. At a later time, a query consisting of a two-tuple, $< Q, t >$, where Q is the query and t is a timestamp, is registered at the information provider. At registration time, $t = -\infty$ and all documents in $\mathcal{D}$ ranked highly relative to the query Q are returned in a search result set $\mathcal{D}'_a$. The current system time, t_a is returned to the user proxy that issued the query. Because the user proxy was told t_a, this value can be used with a timestamp-aware search interface the next time the query is executed at this information provider. At a later time, the user proxy returns with the query that is now $< Q, t_a >$. The query and timestamp are executed in the search engine at time t_b to generate a search result, $\mathcal{D}'_b$ where:

$$\mathcal{D}'_b = \{d_1, d_2, \ldots d_{|\mathcal{D}'_b|}\}, minTimestamp(\mathcal{D}'_b) = i, maxTimestamp(\mathcal{D}'_b) = j, (i > a) \wedge (j \leq b)$$

Assuming append-only information providers as in [TGNO92] and a current time of t_n, $\mathcal{D}'_a \cup \mathcal{D}'_b \cup \ldots \cup \mathcal{D}'_n$ is equivalent to Q executed at t_n without using a timestamp feature. Because the timestamps are monotonically increasing and the query is executed over partitions of the cumulative set of content $\mathcal{D}$, the union of the results contains no duplicated timestamps (and thus no document appeared twice). The search result union contains all documents that could have possibly matched the query at any given time, assuming the information provider loses no documents.

The lossy nature of internet information providers must be taken into consideration as developing the entire persistent query theory around append-only providers would burden the information provider unreasonably and would provide poor service to users. The working assumption here is that providers are "well-behaved," and given working persistent query operation and diligent user proxies, documents exist in $\mathcal{D}$ long enough to be retrieved within reasonable constraints. Thus, the argument above concerning the union of the result sets holds and can possibly increase the number of information items evaluated against a user's query. If Q is executed at the search interface at time t_n, the result set returned for a retrospective query would likely be smaller than that gained by taking the union of the differential search results because of deletions in the content base; at time t_n, the documents that were present during the differential query intervals may have been deleted. Thus, the result of summing the sizes of the partitions over the differential intervals may be larger than the size at the time of a retrospective query or other persistent query evaluation mechanism. Differential querying has the ability to send notifications about documents that would be missed if the provider is not occasionally checked. In addition, the results are delivered to the user proxy as they appear at the provider, thus fulfilling a requirement that the user proxy (and transitively the user) stay continually apprised of new content at a provider. If user proxies are diligent in acting on this goal, they will fetch useful information from a provider "soon" after it is inserted. Thus, with well-behaved providers, user proxies may often obtain content before it disappears. In the case of persistent queries that will be posed in the future and miss previously deleted documents, this is

an unavoidable consequence that would require significant machinery, especially caching, to mitigate. Remember, however, that this is an SDI system which focuses on the current and future information flowing into the information providers. Given an appropriate persistent query mechanism and diligent user proxies, the goals of SDI can still be met even if information providers are lossy in the long term.

The absence of timestamps has a significant impact in the provider's ability to perform persistent querying because of the requirement that each persistent query provide duplicate-free search results to a user. Without timestamps, a complete list of all documents that have been returned by each query must be kept at some location in the system, either at the provider or on the user proxy. Accomplishing this requires retrofitting a total ordering onto each individual's search results for all of their persistent queries. This total ordering can be created by providing a unique, monotonically increasing document identifier for every document returned to a user and filtering new search results against all previous results.

There are several issues with retrofitting timestamps in such a manner. Assuming the processing and storage of the search result lists can be performed on the information provider, the list of document handles in a search result for any of a user's persistent queries is maintained on the provider for every registered persistent query. After a user's persistent query executes on the information provider's search interface, the search result list is filtered against the previous search result list to remove their overlap. Any remaining entries are returned to the user as the "current" search result. If this is the case, the storage requirements for maintaining the list of previously seen information item identifiers for every persistent query would expand quickly and would not scale well for large registered persistent query sets, even in a capable database. In addition, while the computational complexity for differencing the current query results against the previous results are insignificant for a single user, scaling this operation to internet-scale would be a significant computational burden for information providers. Second, if the user proxy maintains the previously seen document list, the same difficulties exist but without the scaling requirement. The storage of the previous search result list is an issue as the previous search result

list grows over time. The process of differencing the lists of current results versus previous results could be less of an issue at the user proxy because of the potentially distributed nature of user proxies; each user proxy may leverage the computational capabilities at its host server. Note, however, that the network load for transmitting the entire contents of a search result list to every user proxy would greatly increase bandwidth utilization.

The presence of timestamps coupled with a search interface and appropriate syntax to facilitate using the two together for searching may be implemented using a method similar to the interface in Figure 3.4.

```
public interface SearchInterface {
  Document[ ] giveAllDocumentsSince( Query q, Timestamp t );
}
```

Figure 3.4: A search interface with timestamp support.

An interface of this style simplifies the process of querying for both the user proxy and the information provider. The user proxy is assured that no results returned overlap in terms of their sequence numbers with the results that the user proxy has previously seen, and the provider does less work in executing the query.

The tradeoffs in terms of providing timestamps are significant. On the one hand, requiring an information provider to create timestamps and have an interface through which to meaningfully use them is a burden placed on providers in a persistent query system. On the other hand, retrofitting timestamp functionality via previous search result list storage and list differencing will force a significant amount of functionality for duplicate search result removal elsewhere in the SDI system. Implementing a search or browse interface that uses timestamps is a significant step to implementing an efficient persistent query system.

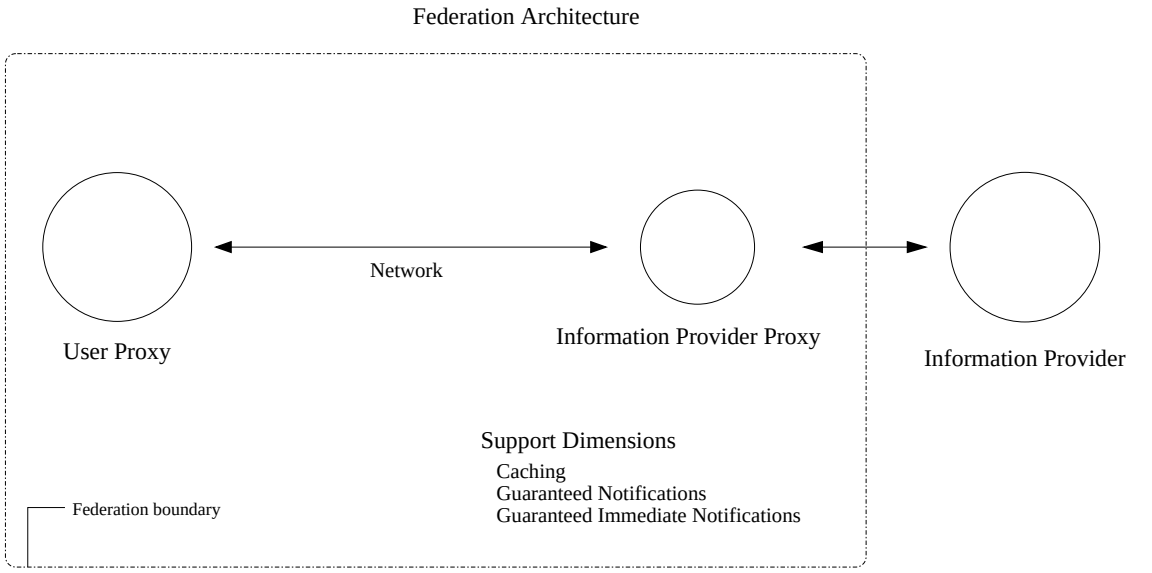

Figure 3.5: Dimensions for supporting persistent queries in a federation

3.4 Federation Infrastructure Requirements

In addition to the requirements placed on information providers to enhance their support for persistent queries, creation of a federation infrastructure can facilitate additional enhancements and can make guarantees that are outside the reasonable scope of the providers. The major difference in requirements between information provider features and federation features is the addition of a system wide infrastructure and object model. This architecture is presented in Figure 3.5; compare this with the simple architecture in Figure 3.1.

The participants in the above model include a new object, the information provider proxy. The provider proxy emulates functionality of the actual information provider, including the publish / subscribe pattern provided by the information provider itself that allows user proxies to register an interest in receiving change notifications from the provider via the provider's proxy. In the federation if notifications are emitted by the information provider, then the provider proxy would be the only subscriber in the actual information provider's subscriber set, and the provider proxy would assume all of the responsibilities

for receiving the notifications and then propagating them along to subscribed user proxies in the federation. In addition, the information provider proxy will export the information provider's search interface, or the provider proxy may provide such an interface if one does not exist on the actual provider. Note that these changes have no effect on the information provider, which behaves exactly the same as if its proxy did not exist. Figure 3.5 will evolve depending on the support dimensions implemented at the federation level.

3.4.1 Caching

Caching is the process of storing a document and its constituent parts for retrieval and use at a later date. In particular, we are concerned with the process of caching documents outside of information providers to mitigate the risk of their deletion at their home provider. Caching is an important process that enhances the reliability afforded to the user because documents in volatile information providers can persist until a user processes a document. While we assume "well-behaved" information providers, a theoretic caching model is worthy of discussion as a dimension in the infrastructure. Particularly in an environment such as the web, caching is necessary both for fault tolerance purposes, in case its server is down and a document is unavailable, and because the content itself might disappear from the provider before the user has the ability to read and deliberately store the document.

Absence of a cache in persistent query system may have an impact on the availability of documents from some information providers. Without caching, the document pointers returned to users through the regular SDI operation may be null pointers by the time the user takes the opportunity to retrieve them. Thus, for lossy providers, caching at some level, for example in the provider proxy or user proxy, will increase a user's likelihood of being able to view a document, but it will not increase the accuracy of the search results delivered to a notification subscriber. One factor to consider is that inclusion of a variety of information providers will mean that different document lifetime policies may be employed at each.

Inclusion of a cache in the infrastructure can be done in at least three locations, the

information provider proxy, the user proxy, and at a third party. Which one or combination is selected depends on policies employed in the infrastructure design. We will consider each in turn. In the case where caching is performed in the information provider proxy, the provider proxy effectively becomes a mirror of the provider itself. The provider proxy must give a location and identifier under which to store every document that arrives in the provider, and in addition, the provider proxy must somehow determine all of the documents that are added each time the provider proxy receives a change notification or polls the provider. Keeping the provider and its proxy synchronized without the cooperation of the provider in divulging new documents will be difficult. Also, the storage requirements in this naive solution will be large. Assuming persistent queries are registered at information providers, an optimization is available by using a table with dimensions:

$$number\ of\ documents\ X\ number\ of\ registered\ queries$$

The table contains entries marking whether or not the $< subscriber\_id, query\_id >$ pair has seen the document referenced by the document identifier dimension of the table. A table entry remains empty until a query is executed against a document set that contains the document identifier associated with the entry. Then, once the search results are returned and the user proxy requests a set of documents, the entry for each document in the document set over which the query was posed is marked. When an entire row is marked, the document corresponding to the row has been a candidate for selection with every query presently registered, and it can be deleted if necessary. Clearly, this protocol is lossy and would not perform as well as keeping all of the documents, but with churning information providers and space requirements, it is a possible solution. One significant difficulty is that queries registered after documents have been deleted will never be able to see the missing documents. As discussed with timestamps in Section 3.3.3, however, the purpose of the persistent query functionality is SDI, which is most interested in the current and future documents. The table itself will be expensive to maintain and manipulate for large systems; if the provider proxy caches, caching all documents would provide better performance.

Caching on behalf of the user in the user proxy will require significant commitment in terms of storage, but the requirement is more reasonable if all of the content does not need to be stored. From an implementation perspective and given the potentially distributed nature of the system, caching on the user proxy makes more sense than caching on the provider proxy for the same reason a query should be kept at the user proxy; see Section 3.3.2. The user proxy is a better judge of the merit gained by caching a document than is the provider proxy, and the provider proxy will not have to store all documents from the actual provider. Because of the well-behaved nature of information providers, a user proxy's ability to respond to a notification quickly and receive a search result translates into its ability to immediately turn around and resolve document pointers in the search result; thus, the user proxy is able to cache exactly what it deems most important to the user it represents.

Caching could also be done in a third party service that is bolted onto the user proxy or is hosted remotely. A third party cache could interact with the information provider, its proxy, or the user proxy, but given the previous argument, it makes the most sense for a cache simply to deal with a user level component. In an ideal scenario, the user proxy receives a change notification, responds by issuing the persistent query at the provider proxy, decides which document handles look the most promising out of the search result, and passes these handles to the remote cache, which will perform the necessary operations to save the documents for the user. This could even be a fee-based service that user proxies may or may not use depending on user preference. Regardless, caching makes sense when considered in the context of a federation, and adding one may add value to users' search results. Because caching takes place after notifications, though, it is presented here for completeness in terms of persistent query support and is outside the scope of the rest of this work.

3.4.2 Guaranteeing Notification Delivery

Guaranteed notifications are a federation-level support dimension that guarantees the delivery of change notifications to a provider's subscribers. This is a further enhancement to the unreliable change notifications described in Section 3.3.1. Guaranteeing notification delivery makes the additional provision that all user proxies subscribed to receive change notifications from an information provider will receive all notifications sent to them at some time, given certain operating conditions. A "guaranteed" notification is always successfully delivered to a user's proxy from an information provider provided the following hold:

- an information provider proxy exists in the federation for the information provider;
- a path through the network exists between the user proxy and the information provider proxy (and vice versa); and
- a user proxy that goes off-line at some time comes back on-line at some later time.

The protocol for delivering a guaranteed notification begins with an information provider propagating a change notification. Unlike change notifications delivered under the simple persistent query architecture in Figure 3.1, this change notification is delivered to an information provider proxy that exists within the federation supporting persistent query functionality. The subscription interface is also implemented at the provider proxy to facilitate change notification subscriptions from user proxies in the federation; the provider's proxy is the only federation participant subscribed for change notifications directly from the actual information provider (which exists outside the federation). The information provider's proxy then delivers the change notification under the model selected for change notification delivery[7]. Then, the provider's proxy attempts to deliver the change notification to every element, $< subscriber\_id, query\_id >$ in the provider proxy's subscriber set. When a user proxy receives the notification, it acknowledges with a successful return value or the absence of an exception. In the event of a failure, the "guarantee" provided by guaranteed notifications is enacted for a the $< subscriber\_id, query\_id >$ pair. The provider's proxy queues

[7] These models, for delivery-per-change, variable rate, etc. were described in Section 3.3.1.

the failed $< subscriber\_id, query\_id >$ pair into a list of other such failed pairs. Because all change notifications from an information provider contain no state and denote a simple, generic event, the guarantee is met with the successful delivery of a change notification to the user proxy in any order. In a simple implementation, the provider's proxy attempts to send a notification to each failed $< subscriber\_id, query\_id >$ pair using a round-robin algorithm that is executed on some time interval, for example daily. When a notification is delivered for a pair, the pair is dequeued[8].

Under these circumstances, a notification that is emitted by the information provider will arrive, at some time in the future, at the user proxy; we call this "eventual semantics" for notification delivery because every pair will *eventually* receive every change notification sent by some information provider's proxy in the federation. The conditions are necessary because the information provider / provider proxy duo can guarantee due diligence in attempting to deliver a notification but can not guarantee its receipt; the conditions guarantee the receipt of the notification. Also, requirements on user proxies concerning finite absences from the system are necessary to prevent an infinitely long presence of a failed notification in the failed notification set. By adding these three conditions, the receipt of the said notification is guaranteed to occur under eventual semantics. Of course, as a practical matter, the user's proxy is also required to unregister from the information provider proxy's subscription interface for all providers that exist in the user proxy's provider set. This is necessary to ensure that the provider's proxy does not waste resources attempting to send a notification for a persistent query or user proxy that no longer exists[9].

Absence of guaranteed notifications will leave user proxies with the same level of support for a persistent query mechanism that exists with unguaranteed, variable rate notifications as discussed in Section 3.3.1. At the propagation of a notification, the user proxy may or may not receive the message. In the case that the message is not received, the information

[8] As a performance optimization, if a $< subscriber\_id, query\_id >$ pair fails, subsequent attempts by the information provider to deliver a notification to the same subscriber for different queries could be suppressed in the current cycle.

[9] Detection of user proxies that violate this optimization can be performed by checking on the continued existence of the user proxy in the federation by pinging or other means initiated by information provider proxies.

provider will notice the fault because the receipt acknowledgement will not return and the provider proxy will not mark the notification as pending in the provider proxy's subscriber set data structure (because the notification is not pending if it was not received). Thus, when some notification is successfully propagated to previously failed user proxy, the acknowledgement will be made and the pending flag will be set in the provider proxy's data structure. Then, using the variable rate notification model, the provider will avoid sending subsequent notifications to the user proxy until the user proxy responds to the notification and executes its query on the provider's changed content. No active attempt is made to re-deliver a single notification. While this difference from guaranteed notifications is subtle, it is important in guaranteeing aspects of a useful SDI systems that works to successfully deliver *new* content to users in a timely fashion; guaranteed notifications facilitate this goal.

Guaranteed notifications allow us to revisit the different decision criteria for sending a change notification. Effective models for implementing change notifications, variable rate, timed, and so on, have already been discussed in Section 3.3.1, but by guaranteeing that a user proxy will see all of its change notifications, the user proxy is able to implement more accurate decision criterion for responding to a notification. For example, the user proxy is able to count the number of documents that have arrived at an information provider and can accurately execute a query after the n^{th} notification arrives. This decision criteria is implemented at the user proxy level and requires no knowledge by the provider or provider's proxy of the user proxy's actions. With guaranteed notifications, no changes are necessary elsewhere in the system to provide user proxies this ability, and implementing the feature here also reduces coupling between the information provider and the user proxy, leaving the provider and provider's proxy entirely genericized.

Implications of supporting guaranteed notifications are significant. The information provider proxy appearing in the Figure 3.5 is necessary because functionality outside the realm of a provider is required. The machinery to ensure that all user proxies receive all notifications is significant and will consume computational resources at the provider proxy. This model is similar to a scenario where the postman leaves mail in a mailbox expecting

someone to retrieve the mail at some time in the future. If the user proxy were to disappear from the infrastructure for some reason, the provider's proxy will essentially "hold the mail" until such time that the user proxy successfully starts receiving notifications again. In addition to requirements on the provider proxy, the user proxy must itself employ techniques that will keep it alive and responsive to heard notifications emitted from the provider proxy. Fault tolerance of such software systems is addressed in other fields of active research. The provider / provider proxy combination makes a best effort to deliver notifications under the conditions specified but cannot ensure the timely receipt of a change notification. Guaranteed notifications must exist in the federation infrastructure level because of the requirement that the information provider not be unduly burdened in supporting persistent queries.

3.4.3 Immediate Notification Delivery

A guaranteed immediate notification builds on the provisions and functionality of a guaranteed notification and makes stronger assurances about when a subscriber will receive a notification. The immediacy of the guarantee is qualitative and states that a user proxy subscribed to receive notifications from an information provider will receive change notifications without significant waiting given three simple criteria (note the similarity to the list of conditions for guaranteed notifications):

- an information provider proxy exists in the federation for the information provider;
- a path through the network exists between the user proxy and the provider proxy (and vice versa); and
- provided the previous condition, there exists a method for contacting a user proxy that is not currently active in the federation, assuming the contact entity is functioning.

The protocol for immediate guaranteed notification delivery is similar to that for guaranteed delivery; the information provider propagates a notification to its provider proxy

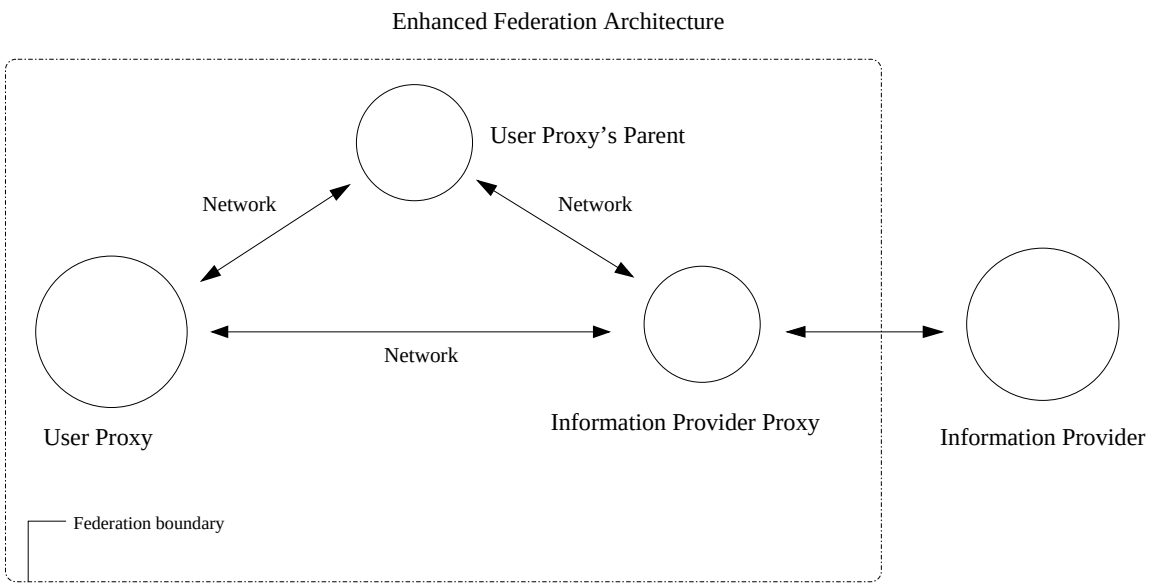

Figure 3.6: Guaranteed immediate notification participants.

which then propagates it to the user proxies in the provider proxy's subscriber set. If delivery is successful, the user proxy can decide whether or not to return to issue the persistent query. If delivery is unsuccessful, the provider proxy must be pro-active in delivering the notification to the user proxy; this requires additional components in the federation infrastructure that can be seen in Figure 3.6.

The user proxy's parent component is contacted by the provider proxy to start or awaken the user proxy via some mechanism. Once the user proxy is executing properly, the notification can be delivered to the user proxy and the appropriate acknowledgement sent back to the information provider proxy. The time delay for delivering an initially failed notification varies depending on the delays in starting the necessary system components, but assuming the conditions hold (except for the liveness requirement on the user proxy), the delivery delay is simply the time required to start the user proxy. A detailed method for contacting a user proxy that is inactive in the federation will be concretely described in Chapter 5.

The absence of guaranteed immediate notifications has the same types of implications as the absence of guaranteed notifications in Section 3.4.2. Not providing notifications at this level of guarantee simply means that user proxies will have to forego reliable delivery of notifications and implement error handling routines, such as query upon failure recovery, in supporting a user's persistent query activities. In addition, more statistics for making judgments about rates of content evolution at an information provider will be missing, impacting the accuracy of timer-based and counter-based notification response models implemented at user proxies. Also, user proxies will have to rely on the less frequent but just as effective semantics provided by variable notifications. In either model, the user proxy can still execute a persistent query at will at an information provider proxy.

Guaranteed immediate notifications imply that if the system is working in an appropriate manner, the notification sent by the information provider will be received at a user proxy and will be delayed only by the network transmission time and the overhead in running the notification handler. If all aspects of the system are functioning correctly, guaranteed and guaranteed immediate notification delivery are functionally equivalent because the notification is not delayed en route to the subscribed user proxies. The difference between the two is the operation of the failure mode if the transmission of the notification is unsuccessful. With failure in the former case, the provider proxy works at its leisure to deliver the notification so long as it is eventually delivered to the user proxy as described in Section 3.4.2. In this case, no additional functionality is necessary outside of the failure routines described for the information provider proxy. In the latter case, the immediacy notion of the notification requires additional support to be able to contact the user proxy "immediately." The requirement here is that there be a way to contact a user proxy that is a member of, but is not currently active in, the federation at any given time. This functionality may exist in Figure 3.6 at one of two locations, at the information provider proxy or at the user proxy.

Immediate notifications can be used to deliver information to user proxies and users in near real time, and given an accurate mechanism for its judgment could facilitate attaching

importance to the delivery of an information item and cause its immediate delivery via a pager, a cellular phone, a PDA, e-mail, or an alarm clock. The judgment of this importance is outside the scope of this work and is left as an information retrieval research question. The immediate guaranteed notification says nothing, however, about the response time of the query interface or availability of the information provider and simply guarantees that the notification arrives at a user proxy in a timely fashion. Our notification models guarantee the delivery of change notifications but can not force action on such messages on the part of user proxies.

3.5 Differing Levels of Support

The beauty of encapsulating an information provider in a provider proxy lies in its elegance and simplicity and decouples the provider from interested user proxies. Support provided by the information provider becomes less of an issue in terms of the features that the provider's administrative authority implement and more an issue of what support can be provided to user proxies in a federation by the provider proxy. In the federation, the interface to an information provider can be genericized. The success of this design is that information providers can seamlessly (but not orthogonally) give differing levels of support to user proxies. A user proxy is able to listen for change events from a provider proxy that sends guaranteed notifications as easily as a provider proxy delivering unguaranteed, variable rate notifications. The guarantees provided by both will differ in their stringency, but the methods of interaction will be the same. This lends a great deal of flexibility to an implementation because of the homogeneous appearance of information providers when viewed from within the federation; however, differing support from provider proxies is not orthogonal because all have the same appearance but are used differently. Thus, care must be taken to denote differences between providers exporting heterogeneous functionality through a homogeneous interface, but in terms of an implementation, this simplicity can be a step that reduces complexity. Because of the orthogonality argument, a federation

should be used to provide the same level of support across all information providers in terms of guarantees made for notifications so that all user proxies expect and receive the same notification behavior from all information providers.

3.6 Conclusion

A persistent query mechanism is one model for implementing an SDI based information distribution system. This chapter has presented the theory behind dimensions of support provided for a persistent query system. Those considering implementing such an SDI system should consider the requirements and burdens placed on components of the system, especially at the information providers. Virtually all of these features can be implemented independently of each other to provide some level of additional support for a persistent query mechanism; however, implementing some of them together can facilitate great gains in SDI system effectiveness, both in terms of results for users and resource consumption.

In implementing a concrete persistent query system, we have selected several of these dimensions. Our system uses a federation infrastructure and provides notifications, a search interface that is coupled with monotonically increasing document identifiers, and the fault tolerance levels of guaranteed immediate resource change notifications transmitted on a variable delivery rate. The design of our persistent query protocol is presented in the next chapter.

4

Protocol

4.1 Overview

Having discussed the various dimensions of support for implementing persistent query functionality in an SDI system, it is now possible to describe the protocol that we have developed to support persistent querying. In order to support a useful system that allows the number of information providers and user proxies to scale, our persistent query protocol requires a support infrastructure of the type described in Section 3.4. An infrastructure will help the system scale and provide additional services to users. Our design goals are those stated in Chapter 1:

- create a lightweight protocol that can be used to disseminate timely information to users with little user effort;
- do not overly burden information providers;
- make the solution efficient in terms of network traffic and computational overhead;
- make the solution scalable; and
- provide a solution that is applicable to the WWW environment.

This chapter details our selection of features to require both at participating information providers and in the federation supporting persistent querying. Justification for each

choice is provided; outstanding issues with the protocol that need to be addressed in an implementation are also mentioned.

4.2 The Protocol

The design of the persistent query protocol follows from the examination of the dimensions of support presented in Chapter 3. Clearly, some of the support dimensions are straightforward and others are more intricate to provide, such as a search interface and caching respectively; these requirements have influenced the choice of features for our protocol. Our basic design is as follows; users pose persistent (and retrospective) queries through a client-side user interface to the user's proxy object, which operates in the infrastructure supporting our SDI system and persists after the user logs out of the client. When the query is posed, the user's user proxy subscribes to change notifications from each of the information providers in the user's information provider set; we allow users to customize items in this set. For scalability and to minimize requirements on information providers, we expect that each information provider is enclosed in a wrapper that is aware of our SDI infrastructure and supports the persistent query mechanism of the system, this wrapper is the information provider proxy or provider proxy for short. No communication between the information provider, provider proxy, or user proxy occurs until a change is detected, but at the time of a change, the information provider sends a simple change notification to its proxy. In the most fundamental action of the protocol, the provider proxy passes the notification along to every user proxy that has registered a persistent query against the changed information provider. User proxies are then aware of a change in the information provider's content. The user proxy is now in control of deciding when to respond to the change notification received from the changed information provider's proxy. When the user proxy decides to respond to the change notification, the results of this query will consist only of the *new* information that has arrived at the information provider since the last time the user proxy posed the query there. The time that the query completes will

be returned to the user proxy along with the search result; the user proxy will use this timestamp the next time the user proxy issues the query to the provider proxy in order to partition the documents at the information provider into the previously queried and yet to be queried sets described in Section 3.3.3. The search result may consist of documents or pointers to documents. A user's persistent query remains registered at all of the information providers until the user either deletes the persistent query or removes an information provider from the information provider set. The notification paradigm used is the variable rate notification model, and those notifications are guaranteed to reach the user proxy under the appropriate conditions. Under these conditions, the protocol supports delivery of notifications to the user proxy in either guaranteed or immediately guaranteed mode, modeling the mailbox or hand delivery described in Section 3.4.2 and Section 3.4.3. Which method is used could be selected by the user, and either method can be implemented by the infrastructure. We have chosen to implement the latter because the former follows naturally from doing so. We do not implement caching and can emulate timestamps and a search interface at the information providers, though provision of them by the information provider would be helpful and would reduce the cost of implementing and maintaining an information provider's proxy.

This protocol is different from many of the previous SDI systems of the past in several ways. First, we believe change notifications are the fundamental component of an efficient persistent query system. We require this feature of the information providers participating in our system. It is possible to implement a wrapper that generates a change notification by polling the information provider, but we are not directly supporting this class of information providers. Change notifications significantly reduce the burden on network and processing infrastructure and yield a significant gain in efficiency. Significant research has been conducted in the fields of distributed event notification services as described by Hinze *et al.*[HF99], Carzaniga *et al.*[CRW98], and Rosenblum *et al.*[RW97], but few information retrieval systems have been implemented on top of such an infrastructure[1]. In addition, our

[1]The MediAS system has been implemented in a digital library environment on top of infrastructure described by Hinze *et al.*[HF99].

notification scheme leaves the decision of whether to respond to a change notification up to user proxies; a notification does not necessarily trigger a flurry of query processing by all of the notified objects. Second, queries in the persistent query protocol are not stored at the information providers. They are kept in each individual user's user proxy object in the SDI system. This yields gains in terms of efficiency for the information provider or its wrapper because they need only know about the location of the user proxies that are subscribed to receive notifications. In addition, we believe given our current implementation that this solution will scale well in an environment of millions of information providers and users. Systems providing information retrieval capabilities to users as opposed to information filtering capabilities have not scaled to such levels in the past.

4.2.1 Necessary and Sufficient Support

Sending change notifications is the fundamental requirement for implementing our persistent query model. Transmission of notifications facilitates the eventual semantics and the variable rate notification paradigms in the system. Both of these reduce the resource requirements in terms of network and processing that would be necessary using other models such as notify-per-change. Alternatives to variable notifications would trade-off between propagating notifications for every information provider change and propagating too few changes to keep user proxies informed. Variable rate notifications allow user proxies to be fully informed if they choose to be so; a user proxy's choice to be fully informed is made by responding to change notifications. This option is facilitated by requiring emission of notifications from information providers.

Notifications are the single sufficient requirement for accurately providing persistent query support. All other functionality including timestamps, a search interface, and the other information provider level support dimensions, can be emulated in the information provider proxy, albeit at significant cost. If polling is permitted as a means of detecting changes in an information provider, even notifications are not necessary to support persistent querying. Many SDI systems in the past have used polling, often including a user

specified interval at which to check for new information items, to detect changes in content at an information provider. This is true even in systems where providers are cooperative with the goals of the system, for example, Pasadena [WF89], Tapestry [GNOT92], and OpenCQ [LPT99a]. Polling is, however, expensive in terms of computational resources. This expense is manifested on the network because of information provider proxies that must poll uncooperative information providers. In order to provide a sufficiently small granularity of notifications to notice discrete changes as opposed to gross changes in an information provider's content, the polling interval must be small. This increases the resource requirements of a persistent query system even more. A larger issue, however, is that by providing notifications, a content provider is in control of the amount of traffic that it receives from interested user proxies; resources are not wasted responding to frequent but non-productive polls. Implementing functionality that prevents an SDI system from polling information providers for changes will likely increase the participation of information providers in the system.

Polling disallowed, change notifications are a necessary and sufficient condition for supporting a persistent query mechanism. This is because in a persistent query implementation of an SDI system, changes must be detected at information providers in order to disseminate new content to participating users. Excluding polling requires a mechanism of some sort to detect changes from at information providers, which leaves the provider to signal changes itself. A change notification implements this mechanism and is a necessary component for a persistent query system. Furthermore, all of the other features can be emulated in proxies or at other locations in the system. A search interface is not necessary because if the information provider is browsable, a provider proxy can turn that browsable interface into a useful search interface. Timestamps are not necessary because either the user proxy or information provider proxy can keep a running list of which document items a user has previously viewed in order to return fresh results to the user. The other three dimensions are implemented at the federation level and require no support from the information provider at all. Thus, the presence of change notifications is a necessary and sufficient condition for

supporting persistent querying in an SDI system.

4.3 Lightweight

We have chosen to implement the variable notification model that was described in Chapter 3. This solves the problem of notifying subscribers about a change at the information provider and provides an excellent model on which to implement further functionality.

Efficiency of the persistent query protocol must be considered at all locations within the SDI system. The two most important areas of consideration are the at the information provider proxy and in the communication between a user proxy and information provider proxy. Communication between the user and information provider proxies should be kept to an absolute minimum. This is a strict requirement because of the degree to which the system must scale to support millions of users effectively. By keeping the communication infrequent, the resource requirements of each system component are reduced. The information provider proxy must be capable of transmitting change notifications to all of the user proxies subscribed to receive them. This message is small, and its frequency is reduced to a minimal amount, while keeping the user proxy informed of changes at an information provider, through the use of variable rate notifications. The efficiency of this notification mechanism is that the information provider proxy need only notify the user proxy as often as the user proxy is interested in processing the persistent query at the information provider. All user proxies could in theory immediately respond to all change notifications, but in practice, we believe that this will not be the case because doing so may be expensive or user proxies will find little benefit in such actions. Variable notifications, as shown in Section 3.3.1, keep notification transmission to an absolute minimum while providing the greatest knowledge about information change to the user proxy at the lowest cost. Thus, while variable rate notifications are not as informative as a notification emission per information provider change, they can convey similar information and are more efficient to implement. In addition to the network communication benefits, variable rate notifications

make providing guarantees to user proxies about notification receipt feasible. Consider the situation where guarantees are made to user proxies but where notifications are transmitted with every content change at an information provider. The rate of transmission errors can remain constant, but by sending more notifications, more errors occur and increase the requirements on information provider proxies to handle such issues. With variable rate notifications and the resulting decrease in notification transmission, the commitment to restart all user proxies inactive in the federation (given implementation of immediate guaranteed notifications) is not as great at the information provider proxy.

4.4 Requirements on Information Providers

We are requiring a simple change notification feature from an information provider and the ability to subscribe a user proxy to such functionality. No additional requirements are placed on providers. This is important because providers can participate "as-is." For cooperative information providers, we would prefer a provider to minimally have a search interface. At this level of support, the protocol can use a wrapper to implement timestamps. A fully cooperative information provider will provide notifications, a search interface, and timestamps. To participate in the persistent query system, though, only change notifications are required, and as will be discussed later, this can be provided very cleanly via a generic component that will handle user subscriptions and change notification propagation.

4.5 Scalability

In order to operate in an information and user rich environment such as on the internet, any solution to the persistent querying problem must be highly scalable. The information provider / user proxy model lends itself naturally to distributed computing, and we will leverage this characteristic. Our idea is to group information providers and user proxies as is convenient for those responsible for administering the servers; for example, each department at a university may wrap its own information providers and facilitate its own members'

participation in a persistent query system. A name service of some sort will process queries to discover information providers at the level of the entire SDI system; scalable information provider discovery will be discussed in Section 6.3. Every information provider's proxy and user's proxy in the persistent query system will have its own unique identifier; each identifier encodes the location of the object on the internet and in the federation system.

4.6 Dynamic Information Provider Environment

Given the explosive growth of the internet in recent years and the number of information providers participating in the medium, it is imperative that a persistent query mechanism be able to leverage information providers that are located in this environment. Often in the past, the information provider set of an SDI / persistent query system has been limited and slow to evolve. While arrival of the Information Age has radically changed how such systems are viewed, limitations on the number of available information providers have consistently been issues in this type of SDI system. In the past, SDI systems have been limited to a few citation databases; more recent systems are limited to a centralized Netnews server. Our vision is much lager and includes the possibility of including tens of thousands of providers present on the internet in addition to databases, libraries, and other content producers in a system implementing our persistent query protocol. Thus scalability is a requirement along with easy integration of heterogeneous information providers. The persistent query protocol is not limited to a single provider type; rather, the protocol's use of information provider proxies yields an architecture facilitating limitless numbers of information providers. All that is required to implement such a proxy is to develop the information provider specific features necessary to export, via a homogeneous interface - see Section 3.5, a notification service, a search interface, and metadata information to user proxies within the SDI system. The architecture of the protocol is highly adaptive to the dynamic information provider environment of the internet.

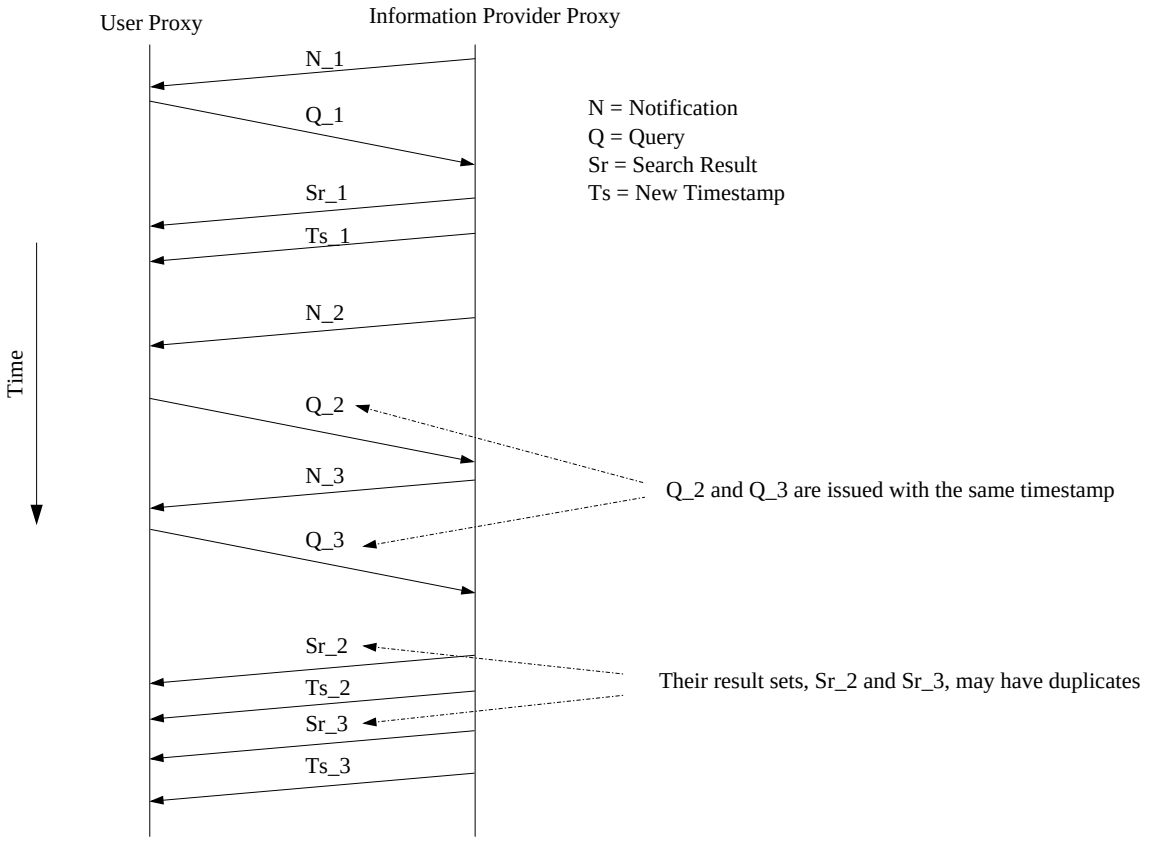

Figure 4.1: Pending notification issue

4.7 A Problem with Notifications

There is an issue with implementing change notifications sent by the information provider to a subscribing user proxy. Consider the following; a user proxy receives a notification from an information provider under the variable notification model and decides to respond to the notification immediately. The user proxy then issues its query to the information provider that has undergone a change, and between the time the query is issued to the provider and before the user proxy receives the search results, the provider proxy propagates another notification. The danger in this situation is illustrated in Figure 4.1.

This is a problem because the two queries issued by the user proxy will have the same

timestamp and will be executing, in general, over the majority of the same content. Assuming that the user proxy attempts to pose the query again (while the previous query is still processing), users may receive duplicated results in their search result lists; this breaks the model described in Section 3.3.3 and enforced by timestamps. This is an unacceptable race condition in the handling of notifications in our SDI system.

The fix for this problem is simple and only requires another flag for every element $l_i \in \mathcal{L}$ in the information provider proxy's subscriber set. The flag is a "notification enabled" flag, and immediately before the user proxy issues a query, the user proxy sets the "notification enabled" flag on the provider to false. Now, no notifications can be sent to the user proxy and the "notification pending" flag can not be marked as true. When the user proxy has finished a query, it re-enables notifications. This is not a problem during query processing because any documents that are received while the query is being executed will be returned to the user proxy regardless of whether or not the user proxy has received a notification for them or not. An issue arises when the query has completed and before notifications are re-enabled because the user proxy will not receive the notification. This can be fixed by propagating notifications, once notifications are turned back on, that were received after query completion but before notifications were re-enabled. Essentially, the disabling and re-enabling of notifications denotes the beginning and end of a transaction between the user proxy and the provider; this keeps each query issuance by the user proxy atomic and simplifies the interaction between the user proxy and the information provider proxy. Disabling notifications raises another simple issue; while the user proxy is disabling notifications before issuing the query, notifications may be sent from the provider proxy before they are disabled. In this situation, the user proxy can behave normally and send additional requests to disable notifications. Through the use of thread safe variables on the information provider proxy side, the "notification enabled" flag will be set once, and using a thread safe variable on the user proxy, the query will only be issued once.

4.8 Conclusion

We believe that the protocol presented in this chapter yields an effective and scalable solution to the persistent query protocol. In addition, the protocol can be applied over a wide variety of information providers and environments. The protocol trades-off notification delivery per change from a provider to all of its subscribers for efficient use of network resources, and this is possible by making guarantees for notifications that are delivered under the eventual semantics / variable rate notification model. We have implemented this protocol in the PIE system [FV99]; this work is presented in the next chapter.

5

Implementation

5.1 Overview

The persistent query protocol may be implemented as a stand-alone application or within an existing system. We choose the latter approach, having already developed a retrospective query system called Personalized Information Environments (PIE) [FV99]. A high-level description of PIE, a description of the test-bed, and a detailed description of PIE are necessary before explaining how the persistent query protocol was implemented in this existing system.

5.2 Concepts of PIE

The PIE system is a software infrastructure for bringing together information providers and users in a user-centric, customizable, and effective search environment with information provider discovery and information retrieval based search tools. PIE provides a palette of information providers to users and lets the user select and interact with individual providers or user-constructed sets of providers. Two basic components comprise the PIE system, PersonalCollection (PeC) and VirtualRepository (VIRP) objects. Both are designed for maximum genericity and flexibility in their roles in the PIE system. In terms of our persistent query discussion so far, the PeC object maps exactly to a user proxy and the

VIRP object maps exactly to an information provider proxy.

The VIRP is a proxy or wrapper for an information provider participating in the PIE system. VIRPs realize an information provider's proxy object. VIRPs are implemented for every type of information provider in the PIE system. In the current PIE implementation, VIRPs and information providers exist in a 1 : 1 relationship. The information provider stores the content and provides a search or browse interface. The VIRP exports the provider's search interface and the provider's metadata to methods that can be used by client objects wishing to interact with the VIRP. This genericity is extremely important when dealing with information providers and is facilitated by leveraging object-oriented programming techniques. Such techniques genericize all specific realizations of a VirtualRepository into an object that can be manipulated by a simple, standard interface. A VirtualRepository base class is subclassed to create instances that wrap information providers of specific types. This allows many different types of objects, including databases, websites, digital libraries, FTP sites, technical report archives, and others to seamlessly participate in PIE with the same homogeneous interface. Currently, one subclass exists, `VirtualRepository_Dienst`, which provides a VIRP wrapper for a Dienst information provider. Additional VIRP types, such as `VirtualRepository_Website` and `VirtualRepository_Library` can be implemented. In order to have a meaningful palette of information providers participating in the system, each provider's cost of participation in PIE must be very low. In PIE, the cost of entry for an information provider is simply to provide a browse or search interface that can be accessed somehow by a VIRP. Including such an information provider in PIE simply requires implementing a VIRP wrapper for the provider. In addition, a VIRP provides a set of metadata about the information provider that it wraps. Currently in PIE, this is a Dublin Core [WKLW98] compliant metadata set, but it may be even more advanced and could include a language model representation of the underlying content indexed at the provider. VIRPs are passive entities from the perspective of users because they provide services only when requests are made to VIRPs by some user.

PersonalCollection objects are constructed and maintained by users of the PIE system and are created by users in order to fulfill a specific information need. In persistent query terms, PersonalCollection objects exist in the same capacity as user proxies. Users explicitly create and name PersonalCollection (PeC) objects; names for PeC objects are unique relative to the PeC set of the user that created the new PeC. A user then adds VIRPs to the PersonalCollection's information provider set. Maintenance of an information provider set is entirely user driven and is a fundamental operation facilitated by a PersonalCollection. The goal of a provider set is to facilitate grouping information providers that have the potential to address the specific information for which the PeC was created. Elements of the provider set can be added or deleted at will. The PersonalCollection object also provides an interface through which the information providers in the PersonalCollection's provider set can be queried. In the basic implementation of PIE, queries are retrospective; they only execute against a collection and return all information items in the collection that match the query at query execution time. Later, we will describe how persistent queries have been implemented in PersonalCollection objects. In naive query processing, a search of the provider set broadcasts the user's query to each information provider, and the PersonalCollection object collects the results of the query and presents them to the user. Currently, the search results are simply concatenated together, but information retrieval techniques may be used at the PersonalCollection to merge and rank the individual search results before presenting them to the user. Items in a search result are pointers to documents, instead of the documents themselves. In our current test-bed, described in Section 5.3, these pointers to documents are URLs. A single user can create an arbitrary number of PersonalCollections, each tailored to meet a different information need. PersonalCollection objects can be shared among users and can exist in different security levels to be accessed by users with the appropriate permissions for a given level. References to all of a user's PersonalCollection objects are stored in the user's UserProfile object in the PIE; the UserProfile is helps to authenticate the user at login time and persistently stores global user characteristics.

5.3 Information Providers in the Test Environment

This thesis is concerned with the use of information providers in the "dynamic web environment." As discussed in Section 2.4, this environment is a difficult operating environment because of its dynamic nature, unreliable communication mechanisms, and the lossy nature of internet information providers. Our persistent query protocol accounts for this by considering these characteristics in the design of the model. We have also attempted to model this reality in how we evolve information providers in our test environment.

The PIE test-bed consists of information providers that run over Networked Computer Science Technical Reference Library (NCSTRL[1]) [DL00] data. NCSTRL is a loosely coupled federation of computer science departments and other organizations that run servers sharing bibliographic data on the technical reports written by each department or organization, called a publisher. Operational details of the NCSTRL network are not of interest here. We use the NCSTRL data that is available via the Dienst[2] protocol. The test-bed is seeded with a large file containing tens of thousands of bibliographic entries, formatted according to RFC 1807[LC98], from over one-hundred thirty[3] different NCSTRL publishers, mostly operated by university computer science departments. The test-bed is created by decomposing these bibliographic entries by publisher. For testing in PIE prior to implementing persistent queries, the decomposed files were indexed in whole to create a collection for every organization; we have written scripts that automate this and the searching processes. In PIE, each organization represents one information provider that is wrapped by a VirtualRepository type subclass called a `VirtualRepository_Dienst`.

For testing purposes, indexing and searching each collection is done locally using a derivative of the WAIS software package [KM91]. Updates to this information can be made at specified intervals or as new documents are found in the bibliographic files available through Dienst sites on the internet. The search interface to an information provider's col-

[1] `http://www.ncstrl.org`

[2] `http://www.cs.cornell.edu/cdlrg/dienst/DienstOverview.htm`

[3] This number is current at the time of this writing (November 2000); the list of current publishers in NCSTRL can be found at `http://cs-tr.cs.cornell.edu/Dienst/UI/2.0/ListPublishers`

lection is provided by freeWAIS and is executed through the `VirtualRepository_Dienst` object that wraps each Dienst publisher. The freeWAIS search results are sent to a text file and are then parsed to create an XML search result that can be returned to the PersonalCollection through which the query was posed; this XML result can be displayed in the client-side user interface. In this interface, each VIRP is displayed as an icon that the user can drag-and-drop into a PersonalCollection's information provider set. The `VirtualRepository_Dienst` wrapper for the NCSTRL test information providers populates a Dublin Core compliant metadata set that describes the provider and content of its underlying collection.

5.4 Implementation of PIE

Now that the major PIE components and the test-bed have been discussed, an explanation of the architecture and implementation of PIE is helpful and builds the foundation for explaining the changes required to implement and analyze persistent queries in PIE. These main components of PIE interact to provide users an infrastructure in which to complete information discovery and retrieval tasks in a user-customized, user-centric environment.

PIE was initially implemented in C++ on the Legion distributed object meta-system described by Grimshaw *et al.*, [LG96, GW96]. The original PIE user interface was a simple command line tool that allowed a user to create, query, and destroy PersonalCollection objects. Retrospective querying was the only type of querying available in this system. All PeC and VIRP objects were named, and the Legion system provided an excellent naming abstraction – a "context space" in which objects could be named, stored, and retrieved. Context space is similar to a file system's directory hierarchy because contexts (directories) can be created, nested, and destroyed. Object instances (files), including PeCs, VIRPs, and UserProfiles, can then be inserted into context space and named with the familiar `root directory/sub-directory/NamedObject` syntax. Clients interacting with the Legion server can then resolve to instances of VIRP and PeC objects by specifying a context

name (path name) and, using a Legion method invocation, can resolve a context name to an object instance. We implemented a Java client-side user interface that made the construction of and interaction with PersonalCollection objects more user friendly than the original command line client. Communication between the Legion server and Java user interface is XML[4] based. For many reasons, this has turned out to be an excellent implementation decision because of the platform neutrality of the XML format, its readability, and ability to transmit entire data structures in the well-supported, simple string type.

The server was ported to Java / CORBA because the resource requirements of Legion were too significant for our intended server infrastructure. CORBA requires a short introduction to provide context for a discussion on the scalability of a PIE system based on CORBA. The fundamental component in CORBA is the Object Request Broker (ORB). The ORB is a communication bus for object interactions, which may take the form of method invocations or message passing. The CORBA specification [Gro00] details the format in which data will be transmitted over IIOP, a TCP/IP based transport protocol that is part of CORBA. This includes standardized formats for basic types such as integer, floating point, and string typed data; in addition, IIOP also details the ways in which user defined objects are transmitted. The first step in defining a CORBA based system is to provide the interface to the CORBA objects in the Interface Definition Language (IDL). The IDL provides a language neutral way to define objects and method signatures that are then mapped to a specific programming language. This mapping is done by an IDL compiler that reads the IDL file to generate, for example, Java stubs and skeleton classes that are used by the client and server respectively to facilitate a client's method invocation on CORBA objects. During a CORBA method invocation, parameters passed to the method invoked on a CORBA object are marshaled on the client side by the stub classes generated above. Marshalling a parameter turns it into the common form that is described in the CORBA specification. These parameters travel through the ORB over IIOP, are unmarshaled on the server-side in IDL generated skeletons, and are passed as parameters to the

[4] `http://www.w3c.org/XML`

method on the object on which the method was invoked; return values are transmitted back to the method invoker in a similar manner. This mechanism allows CORBA to facilitate communication between many different server platforms and CORBA-side programming languages. For example, the method invocation could be performed between a LISP client and a C++ server. This is possible because of the common transport mechanism and through the use of stubs and skeletons. This operation is important in understanding how PIE clients resolve and interact with server-side objects; it is also important to provide flexibility in terms of language and server platform to participants in the PIE system. Additional CORBA information can be found in Henning and Vinoski's excellent book [HV99]. One issue in supporting this functionality is providing a naming convention so all CORBA objects can be accessed by any interested clients

CORBA provides built-in services such as naming, transaction, persistence, messaging, and event notification services which may be useful to supporting persistent querying in PIE in the future. Currently, we use the CORBA Name Service, which is very similar to the Legion context space model, to name and locate server-side PIE objects. The Name Service for an individual CORBA server is obtained by obtaining a handle to the server's ORB; ORBs are located by providing an IP address/DNS name and port number. In PIE, we supplement this with the context name of a server-side object of interest. Our server side CORBA objects are the following:

1. Personal Collection (PeC);
2. Virtual Repository (VIRP);
3. User Profile;
4. Object Server;
5. User Manager;
6. Selection Broker; and

7. Object Monitor.

All of PIE's server side objects, those in the list above, are named with a custom object naming scheme that consists of the following data:

$$< \text{server IP address} : \text{server port number} : \text{context name} >$$

For example, the address:

$$< 127.1.2.3{:}1025{:}/pie/virps/ncstrl.uva\_cs >$$

will locate the University of Virginia, Department of Computer Science technical report NCSTRL VIRP that is running as an object that is registered in context space as `/pie/virps/ncstrl.uva_cs` in the CORBA Name Service that is listening at the IP / port number location `127.1.2.3:1025`. In order to attach to this server object, a client would bind to an ORB located at this address, obtain a handle to the Name Service, request the object by context name using the Name Service, and invoke methods on the object as if the object were running locally using the method signatures defined in the IDL for the CORBA object.

Object names in PIE are created once at the object's home PIE server and persist for the lifetime of the object; we call these names ObjectHandles. A discussion of scaling this naming scheme and of distributed discovery of other PIE objects is provided in Section 6.3. The basic architecture of a single PIE server is shown in Figure 5.1.

The context space locations of objects of each type and their cardinalities at a single PIE server are shown in the Figure 5.1. On a PIE server, there may be many VIRP, UserProfile, and PersonalCollection objects. VIRP objects are administered by some authority and wrap an information provider, and PeCs are created by users who then own them. The first time users log into PIE, they provide a username and a password. This username is used to create a sub-context (or subdirectory) in the */pie/users* context at the server. The UserProfile maintains the user's login name, password, and list of all PersonalCollection objects owned by the user. At each PIE server, there is also one of each of the remaining

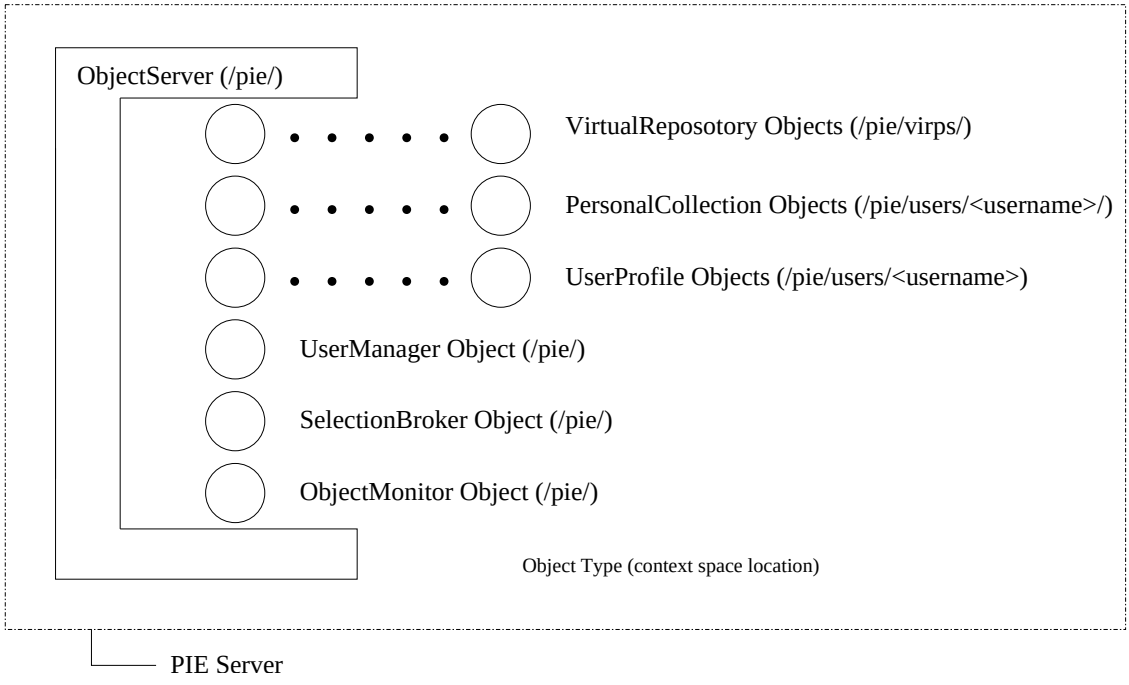

Figure 5.1: Single PIE server architecture.

objects, ObjectServer, UserManager, SelectionBroker, and ObjectMonitor, all of which exist in the */pie* context and consistently named as stated at every PIE server. The ObjectServer is in charge of administering all requests for objects that are stored under the root */pie* context. The UserManager maintains and serves UserProfile objects and handles user log-ins and log-offs. The SelectionBroker provides a service used at query time to select from a PersonalCollection's information provider set the candidate providers most likely to yield information matching a given query based on representations of the content at each provider; it is used for efficiency purposes. Powell [Pow00] describes this process in detail. Finally, the ObjectMonitor runs independently of the other PIE server objects and monitors a PIE server's context space. The monitor keeps track of the objects successfully running in context space and provides an administrator a method for maintaining a real-time view of a single PIE server; the monitor is optional and must be started explicitly by an administrator. When each of these objects is instantiated, it is bound to a context name by invoking a method on the CORBA Name Service running at a PIE server. These objects make up the core of the PIE implementation; each plays a role in servicing user method invocations. In the current PIE implementation, there is a graphical client-side interface and a graphical server-side interface.

5.5 Implementing Persistent Queries in PIE

The PIE system provides an ideal environment in which to implement persistent queries. PIE has an established distributed architecture that decouples the information providers and suppliers and has properties to facilitate scaling to large numbers of participating objects (both PeCs and VIRPs). In addition, PIE natively provides locations to implement functionality needed to support making guarantees about notification delivery as discussed in Section 3.4.3. The mapping of persistent query entities to PIE objects has been mentioned; in persistent query terms, PersonalCollections act as user proxies and VirtualRepositories act as information provider proxies.

The persistent query protocol discussed in Chapter 4 requires changes to the PIE infrastructure and has been realized as discussed in the following sections. The information providers in the test bed, however, are not ideal because they do not provide change notifications or timestamps. The first issue requires polling of providers to determine change, but in our test bed, this is a simple process. In the header for the bibliographic file for each information provider, there is an entry describing the number of documents currently in the publisher's collection. In addition, scripts allow the information providers to have a variable churn rate, so our polling interval and document arrival rates can be varied for testing purposes. The polling code exists outside of the VirtualRepository objects and is implemented as an external observer to the information provider, thus decoupling PIE from polling. VIRP objects wrapping a provider receive a notification from the observer that is watching the information provider for an upward or downward change in the number of documents at the provider. In terms of timestamps, all NCSTRL documents are stamped with a name, but the requirement of a monotonically increasing identifier is not met. Thus, the worst case solution for providing differential querying discussed in Section 3.3.3 is implemented for each persistent query registered at the provider. This does not break the differential querying model presented in Section 3.3.3, it just moves the task of differencing the current search result with the previous search result list from the information provider to the PersonalCollection object. Users still receive only the *new* items that have arrived at an information provider in the search result list for a given persistent query. For every persistent query a user has created, the PersonalCollection owning a persistent query maintains a list of previous search results for the query accumulated over the lifetime of the query. In implementing this functionality at the PersonalCollection level, we have shown that this is a reasonable addition to make for information providers that are uncooperative in this dimension.

Two objects must be modified in the base PIE implementation in order to implement persistent queries, the PersonalCollection and the VirtualRepository. In addition, two data structures must be added, a query execution time table and a persistent query list; these

```
public interface PersonalCollectionClientInterface {

   /* used to register and unregister persistent queries */
   String registerPQ( String persistentQueryXML );
   boolean unregisterPQ( String persistentQueryXML );

   /* used to get queries and query results from the PeC */
   String getPersistentQueryResults( String persistentQueryXML );
   String getPersistentQueryList();

   /* used to monitor the state of persistent queries from a running client interface */
   boolean testPersistentQueryResults( String persistentQueryXML );
   String getChangedPersistentQueries( );
}
```

Figure 5.2: Graphical client $\leftrightarrow$ PersonalCollection interface.

structures are present in the PeC and VIRP objects respectively. Each of these four changes will be discussed in turn.

5.5.1 Personal Collection Changes

To support persistent querying, PersonalCollection objects must export functionality that provides users the ability to pose and delete persistent queries, retrieve the list of current persistent queries, and retrieve the current search result list for a selected persistent query. The interface for this functionality is shown in Figure 5.2.

Note that all input and output data structure types are CORBA strings and use XML as the message format; in reality, these string types are representations of deeply nested data structures that have been linearized into the string representation and are restored to their native types when they reach the caller. For example, the list of persistent queries consists of the objects QuerySet which are n PersistentQuery objects each with a QueryIdenifier object all of which are present within the linear XML structure.

In addition, the PeC object must provide an interface to VIRP objects to be used at notification time. This interface has a method to which notifications are delivered. If a

```
public interface PersonalCollectionVIRP {
  boolean receiveNotification( String notificationXML );
}
```

Figure 5.3: VIRP ↔ PersonalCollection interface.

```
public interface FolderManagerInterface {
  public boolean makeFolder( String folderName );
  public boolean deleteFolder( String folderName );

  public boolean addItemToFolder( String folderName, String documentHandleXML);
  public boolean removeItemFromFolder( String folderName, String
documentHandleXML);

  public boolean testFolderExists( String folderName );

  public String getFolder( String folderName );
}
```

Figure 5.4: FolderManager interface.

notification is received through this interface when a VIRP invokes the method, this meets the "in-hand" delivery of notifications described in Section 3.4.3. The interface is shown in Figure 5.3.

With the addition of persistent query support, PIE evolved to maintain a list of persistent search result folders, similar to those provided for users of e-mail tools. Users can create a folder into which document pointers are stored and all folders are accessible through all PersonalCollection objects; they are stored and managed at the level of users' UserProfile objects. These folders exist in persistent storage on disk and are synchronized to control concurrent accesses to the contents of a given search result list. The interface for interactions with such lists is shown in Figure 5.4.

All PersonalCollection objects have a single FolderManager object that keeps track of search results for each persistent query. The FolderManager implements simple spin locks

per file so that file accesses are synchronized. The PersonalCollection object writes to the search results when new documents that have not previously been received arrive in response to a query at a VIRP, and the PeC object reads from disk when the user requests the contents of a search result file. Because semantics for how the user receives new search results are not provided in the persistent query protocol, access to search results stored on disk must be atomic and controlled elsewhere. Folders are stored in XML, including folders that contain previous search results. Items can be added and removed from the folders through the interface provided by the FolderManager; this gives users the ability to move document pointers from one folder to another. The functionality that we have modeled here is that of user creation and modification of folders in a typical e-mail client. Each user's UserProfile object also has a FolderManager that manages storage and access of document pointers that are stored in folders into which document pointers from any persistent query or PersonalCollection can be placed.

The PersonalCollection object itself must keep track of its persistent queries and the last time they were executed at each of the information providers in the PeC's provider set. This is necessary to implement the differential querying described in Section 3.3.3 because in order to partition the space of documents stored at an information provider, the last query time for each persistent query must be provided to separate the documents into previously seen and not previously seen sets. The QueryExecutionTimeTable is where this information is stored. The table is a hashtable of query identifiers, which are assigned when the query is created by the PersonalCollection object over which the persistent query was posed. Each query identifier points to another hashtable that contains $<$ `key, value` $>$ pairs of the form $<$ `VIRP ObjectHandle, last query time` $>$, for every query identifier. The cardinality of the elements in this latter hashtable are related $1:1$ with the size of the information provider set for the PersonalCollection. When a change notification is received at a PeC for a specific query, the query identifier and VIRP's ObjectHandle are used to hash into these two nested tables to retrieve the last query execution time for the notified query at the notifying VIRP; this time value is then passed to the VIRP at the time of

query issuance. When the query completes on the VIRP, the completion time is sent back to update the time entry in the QueryExecutionTimeTable.

5.5.2 Virtual Repository Changes

Implementing persistent query functionality also requires adding features to the VIRP. The VIRP object must provide a subscription / unsubscription facility for PeCs to express interest in receiving notifications, an interface so that a PeC can enable / disable notifications as necessary, and an interface to the information provider's search capabilities. The subscription / unsubscription interface at a VIRP is used by all PersonalCollection objects that need to receive notifications from the VIRP. PersonalCollections identify themselves during subscription / unsubscription by providing their ObjectHandle and the XML representation of the PersistentQuery object being registered at the VIRP[5]. Enabling and disabling notifications is necessary for the reasons described in Section 4.7. Finally, the VIRP exports the search interface for the information provider in the generic way defined in the VirtualRepository base class. If the information provider has a search interface, the provider proxy's search interface simply act as a pass through and will interact directly with the interface available on the provider. For example, a VIRP wrapping the `http://www.slashdot.org` website would pass any query received at the VIRP directly to the search interface on the web page. Search results are returned to the PersonalCollection by parsing the resulting page for the results of the search at the site. If the information provider does not have a search interface, the VIRP may emulate one internally over the VIRP's representation of the provider's content. Means for accomplishing this are a policy decision left to the information provider's VIRP implementor.

The inheritance hierarchy of the VIRP objects is an important aspect of PIE because significant parts of the persistent query system are common to all of the VIRP objects that are persistent query capable. The inheritance hierarchy is shown in Figure 5.5.

In implementing our design of the persistent query protocol, this hierarchy is impor-

[5]Presently, the text for the query is ignored by the VIRP. Only the persistent query's unique identifier is retained for storage and subsequent use.

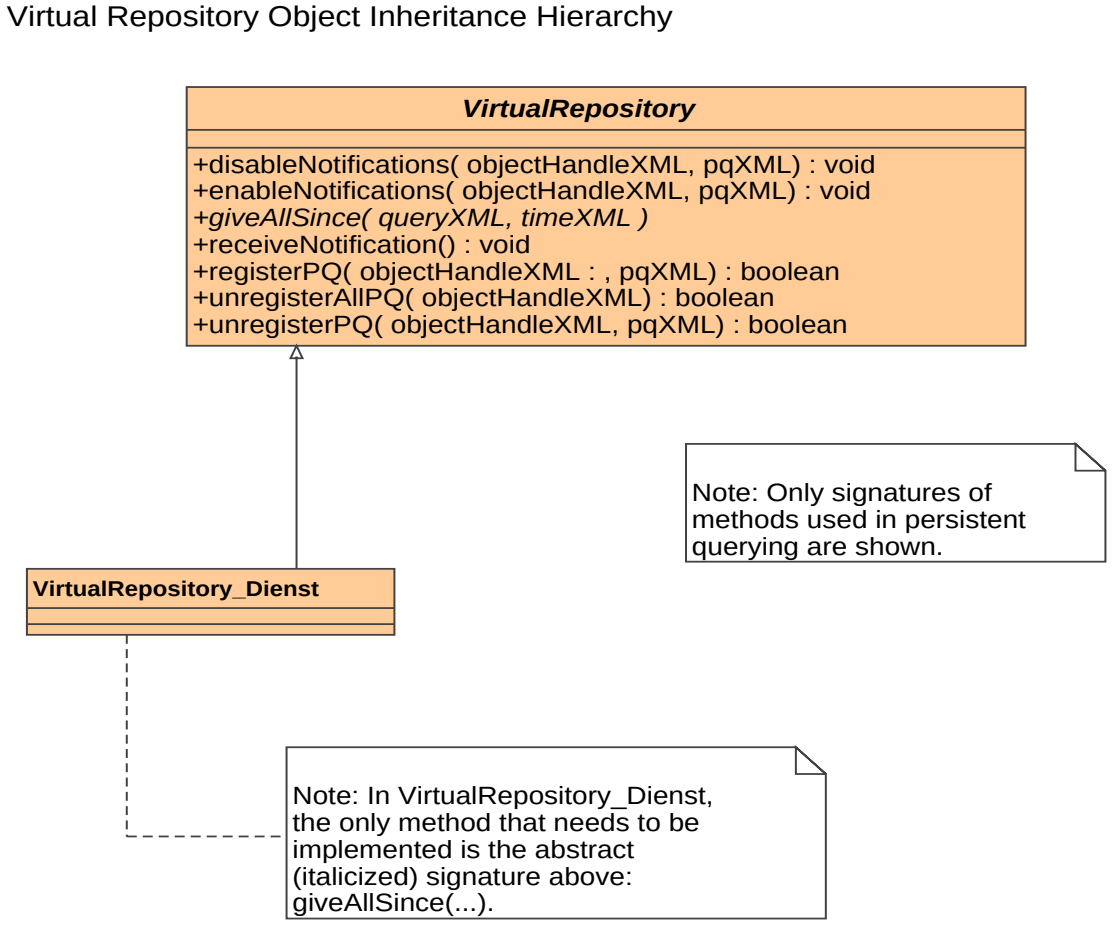

Figure 5.5: VirtualRepository inheritance hierarchy.

tant because much of the functionality required by the VIRP objects is common for all types of VIRPs. For example, managing subscriptions / cancellations and propagation of notifications to user proxies are routines that, under our protocol, will be common to all information provider types. This decomposition of functionality is important because significant development effort does not need to be expended to create VIRP wrappers for additional information provider types; essentially, only the the search interface at a specific information provider must be encapsulated in the VIRP type subclasses implemented to wrap that provider type.

In the VIRP, a data structure must be maintained that describes the list of persistent queries that are registered at a given VIRP. This data structure, the PersistentQueryList, consists of a hashtable with keys that are the ObjectHandle for each PersonalCollection object that has at least one registered query at this VIRP. The value of each key is a data structure with a list of $<$ *query_id, notification pending, notification enabled* $>$ elements. This information fully describes the subscription state of all queries registered at the VIRP by all interested user proxies. When the PersistentQueryList data structure is manipulated in a VIRP object, it is locked so that changes can only be made atomically to its information; this helps to prevent race conditions where the representation of notification state for each $<$ *subscriber_id*, *query_id* $>$ pair.

5.5.3 PIE Infrastructure Changes

In order to implement the guaranteed immediate notifications discussed in Chapter 3, a method must be provided for contacting PersonalCollections that are off-line or inactive in the system at the time of a notification. In addition, when user proxies and information provider proxies are off-line or inactive in the PIE system at a given point, they must persist in a storage medium so that resources at their host servers are conserved.

Recall that inter-object method invocations and messaging in PIE over a CORBA ORB are performed using XML. This was a convenient choice because it does not require CORBA definitions of the many different types of data structures that would be transmitted between

```
public interface Persistent {
   public Persistent(String XML);
   public void parseXML(String xml);
   public String createXML();
}
```

Figure 5.6: Interface for XML based persistence in PIE.

PeC and VIRP such as search results, queries, and other data. All of these objects are passed as XML in the native CORBA string type. Virtually all objects in the PIE system, including those that are not CORBA aware, contain three important methods to support persistence in XML that are shown in Figure 5.6.

The first method is a constructor that receives an XML string as its parameter. After any data structure initializations, this string is passed to the `parseXML()` method which would re-creates all of the persistent data stored in the object and populates any non-primitive type objects that are data members in the class. The `createXML()` method creates a deeply nested XML string that contains all of the data necessary to re-create the object at a later time. This operates similarly to the Java serialization infrastructure but is more effective because it can operate with CORBA and is human readable while other persistence mechanisms are not. Java serializability requires Java remote method invocation instead of CORBA middleware. In addition, as PIE scales and databases large and small are used at servers in lieu of storing persistent data as files on disk, the string type of the representation is simple to implement and does not require BLOB types in the database which would be necessary for serialized Java objects. XML strings are also highly compressible and in addition can be encrypted / decrypted for security during transmission. All of the CORBA aware PIE objects, mentioned previously, can be written to disk in a persistence structure that emulates their locations in the naming context space of the CORBA name service.

The ObjectServer plays a significant role in being able to save and restore CORBA

objects on its own server. The ObjectServer can shutdown parts or all of a running PIE server to be restored at a later date. This is useful if objects need to move from server to server or to conserve server resources by shutting down objects that have not been used recently. For example, in order to shutdown a PersonalCollection type object, the ObjectServer simply must resolve to the name of the object using the CORBA Name Service and execute its `createXML()` method. In fact, an ObjectFactory and PersistenceService have been implemented to abstract the startup and shutdown processes for objects.

The importance of this process is that, given a running ObjectServer, any object at a site can be started locally (by the local ObjectServer) or remotely (by invoking a method in the remote ObjectServer's IDL interface) by another object. This is particularly useful when implementing guaranteed immediate notifications. Consider the following scenario. At notification propagation time, a VIRP (provider proxy) needs to transmit the notification to a PeC (user proxy). If the PeC is not able to receive the notification because it is not running, an exception will be thrown when trying to execute the method from the VIRP on the PersonalCollection. The error handler at the VIRP contacts the ObjectServer owning the PeC to wake the PersonalCollection object up and restore it from disk. Then, the notification can be transmitted from the VIRP to the PeC successfully. If the ObjectServer is not running, there is no way to contact a missing user proxy, and the preconditions for transmitting the notification as described in Section 3.4.3 fail. This latter problem can be solved by implementing the ability to start the ObjectServer remotely from somewhere in the PIE system. This is possible, but currently, we have not implemented this latter functionality.

5.6 The Protocol in PIE

The protocol as implemented in PIE is best viewed using sequence diagrams. These diagrams show the call and return sequences as methods are invoked inter- and intra- object. For clarity, the sequence diagrams here explain the purpose of the method invocations in

plaintext as opposed to simply providing a method signature. Four sequence diagrams are presented that describe the subscription (Figure 5.7), notification handling (Figure 5.8), unsubscription (Figure 5.11), and error handling (Figure 5.9 and Figure 5.10) for persistent querying in PIE. All horizontal lines in the figures represent method invocations; in all cases, return values are transmitted but are generally omitted in these figures. Vertical lines represent the passage of time from top to bottom and invocations between vertical lines occur inter-object. Those invocations starting and ending on the same vertical line (object) are performed intra-object (such as Figure 5.8, Step 2).

Personal Collection objects are the entry point for persistent queries into the PIE system. A user poses a persistent query through a query box in the client-side PIE graphical user interface. The query box is simply a text box for the query and a check box used to signify whether the query is retrospective or persistent. All of the inter-object method invocations occur through a CORBA ORB. The sequence of subsequent actions in the PIE system in response to the user's actions is shown in Figure 5.7.

The user's PersonalCollection receives the query (step 1) and registers the query at each of the VirtualRepository objects that the PersonalCollection has resident in its information provider set at the time the query was issued (step 2). At the time of registration, the query is issued retrospectively at each of the information providers at which the query is registered. When all of the VirtualRepository registrations have completed at the PersonalCollection (after step 7), the collected search results from all of the VirtualRepository objects are returned to the user (step 8). Currently, these results are simply concatenated together but could be merged in the PersonalCollection object. Users are presented with this list of search results in a window on their desktops. Now, users are retrospectively up to date with all of the meaningful information relative to a user's query that was found in each of the information providers in the PersonalCollection object over which the query was posed.

The system is inactive until the content at some information provider changes; at this time, the information provider emits a simple change notification to the provider's VirtualRepository wrapper. This results in a series of method invocations used to notify the

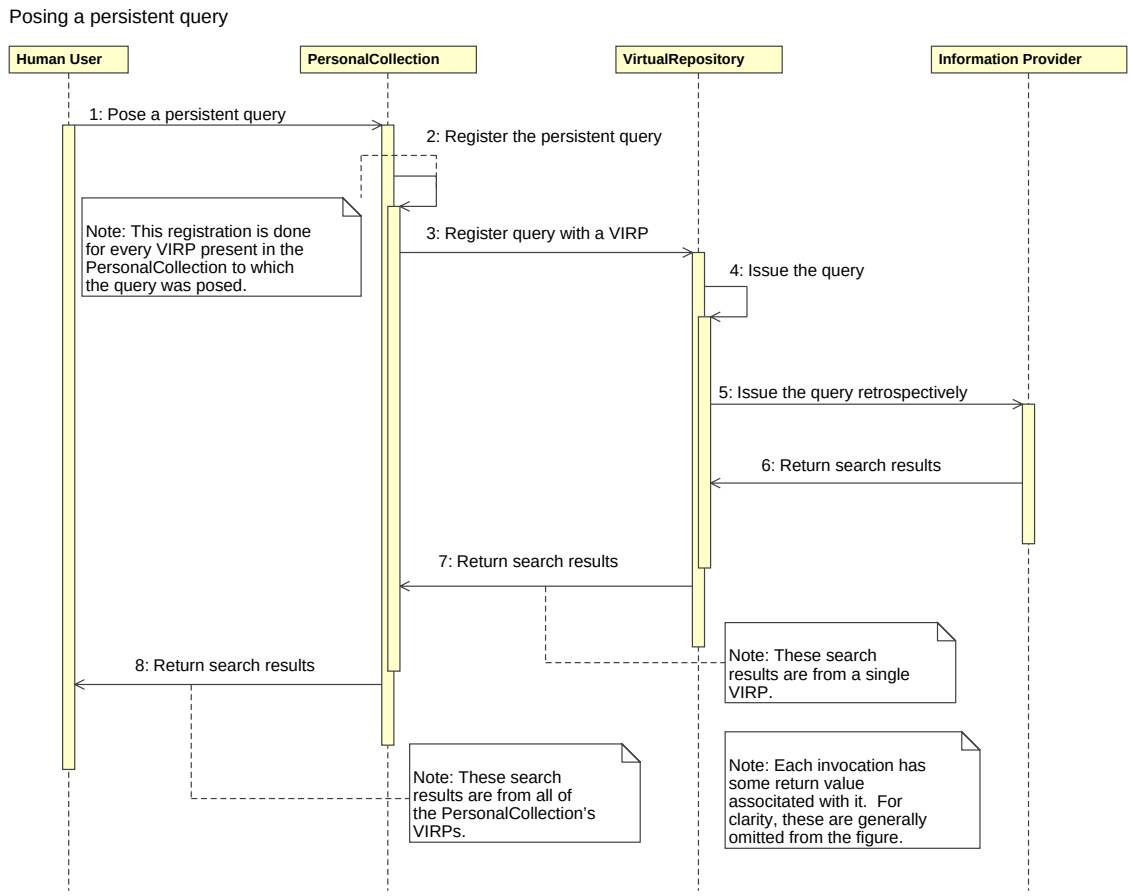

Figure 5.7: Posing a persistent query in PIE.

VIRP's subscribers of a change in the provider's content. These events are sequenced in Figure 5.8.

This series of calls occurs in response to a change notification sent from the information provider to the VIRP signaling a change in the provider's content. The VIRP then takes the notification and propagates it to each of the $< subscriber\_id, query\_id >$ pairs that are registered in its subscriber set. This happens in steps 1 and 2, respectively. Once the notification has been received, the PersonalCollection has the option to respond immediately or to respond later. When the user proxy decides to respond, the protocol execution picks up at step 4. First, as discussed in Section 4.2.1, the PersonalCollection must disable notifications originating from the VIRP in order to prevent the race condition previously described. The acknowledgement of this operation is known to the invoking PersonalCollection object by the absence of an exception during the call and the absence of an error code in the return value. Once this precaution has been taken, the PersonalCollection object issues the query to the VirtualRepository object whose information provider's content changed (step 5). When the PersonalCollection issues the query, the timestamp of the last time the PersonalCollection issued the query to the VIRP that propagated the notification is included; this timestamp is taken from the QueryExecutionTimeTable data structure described earlier. As discussed in Section 3.3.3, this timestamp conveys no notion of system-wide time and only applies to the information provider wrapped by the VIRP. The search is run against the new content at the information provider after step 6 and the search results are returned to the VirtualRepository in step 7. Because every type of information provider wrapped by a VirtualRepository may have a different format for presenting search results, each subtype of VIRP will have a method for parsing search results that is tailored to the format of the search result returned by the provider type. For example, the NCSTRL search result format the test-bed returns is consistent and parsed by a `VirtualRepository_Dienst` type object whereas the `http://www.cnn.com` VIRP will require a search result parser tailored to specially read search results, perhaps formatted in HTML, from that information provider. Once the search results have been parsed, a

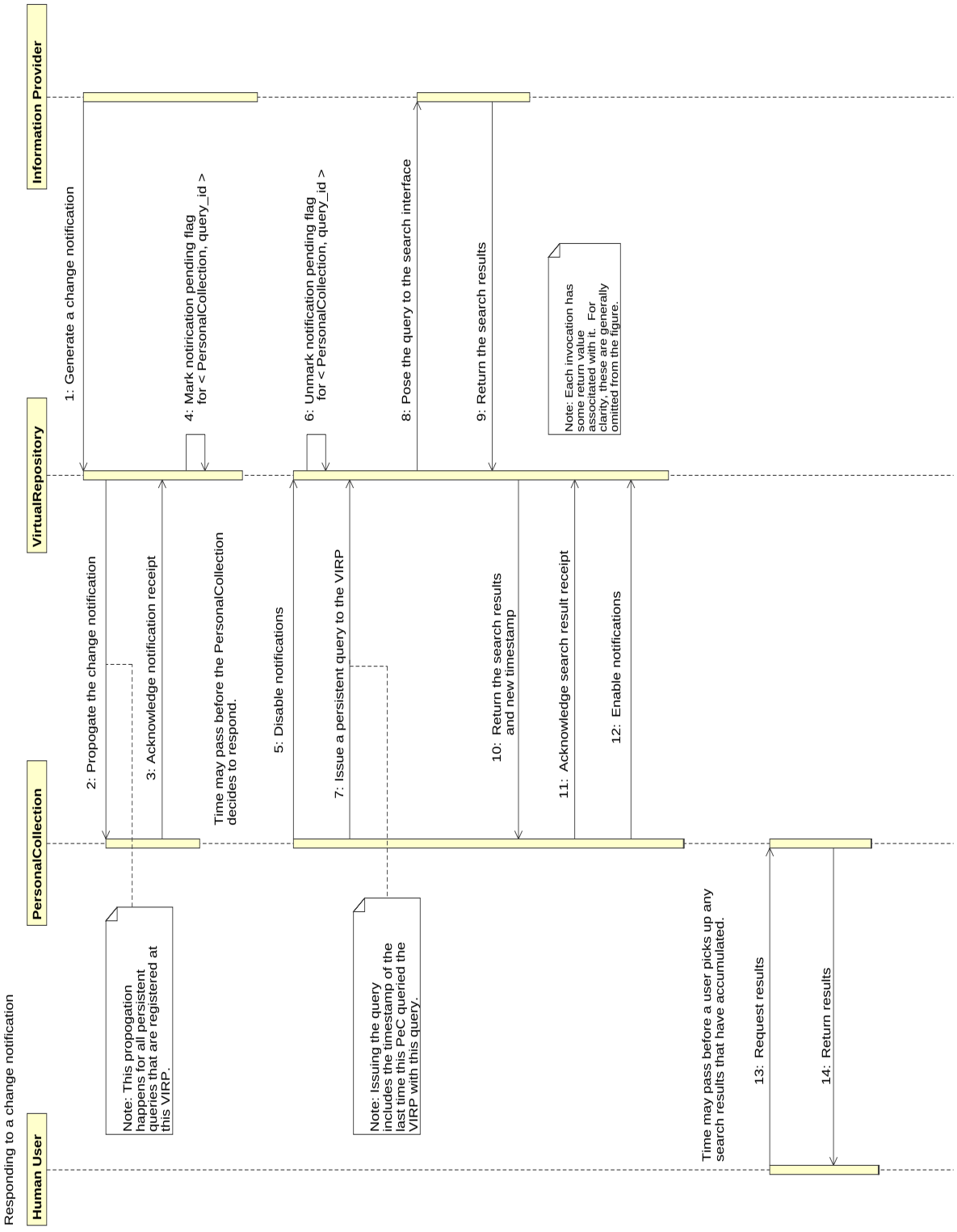

Figure 5.8: Handling change notifications in PIE.

search result set is created on the VIRP and populated with pointers to information items resident at the information provider; this search result is turned into XML and, in step 8, transmitted back across the CORBA bus to the PersonalCollection object that issued the query. At the PersonalCollection object, this search result is appended to a growing list of search results that are pending review by the user and may be differenced against the cumulative previous search results if the information provider does not support searching with timestamps. Search results are stored by the FolderManager in XML on disk or in another persistent storage mechanism. Currently, these search result lists are temporally ordered on a first come, first appended basis, although other presentations including ranking can be provided. When the user next requests this search result, the results are read out of storage and returned to the user's client interface in XML in steps 10 and 11 respectively. The client interface then renders the search result XML into a window where the user can manipulate the results. Further interaction with the search results is similar to reading e-mail; the user can create folders at the UserProfile level that are visible to all PersonalCollections into which to place pointers to information items of interest from any search result at any PersonalCollection. The UserProfile object is responsible for administering thread synchronized access to these folders and for keeping track of their location (usually in the persistent */pie/users/* $<$ *username* $>$ directory on disk) through the FolderManager interface in Figure 5.4.

In addition, there are error recover modes for failed notifications that occur in persistent querying. In the event that a VIRP can not deliver a notification to a PersonalCollection object, the VIRP may contact the ObjectServer at the PersonalCollection's PIE server. One of two error recovery modes can then be employed. The first method is shown in Figure 5.9.

In this method, the ObjectServer can accept the notification from the VIRP and write the notification to a special location that is checked the next time the PersonalCollection object wakes up in step 3. When the PersonalCollection awakens, if there are notifications present in this place, the PersonalCollection is now aware of the changes at the VIRP(s)

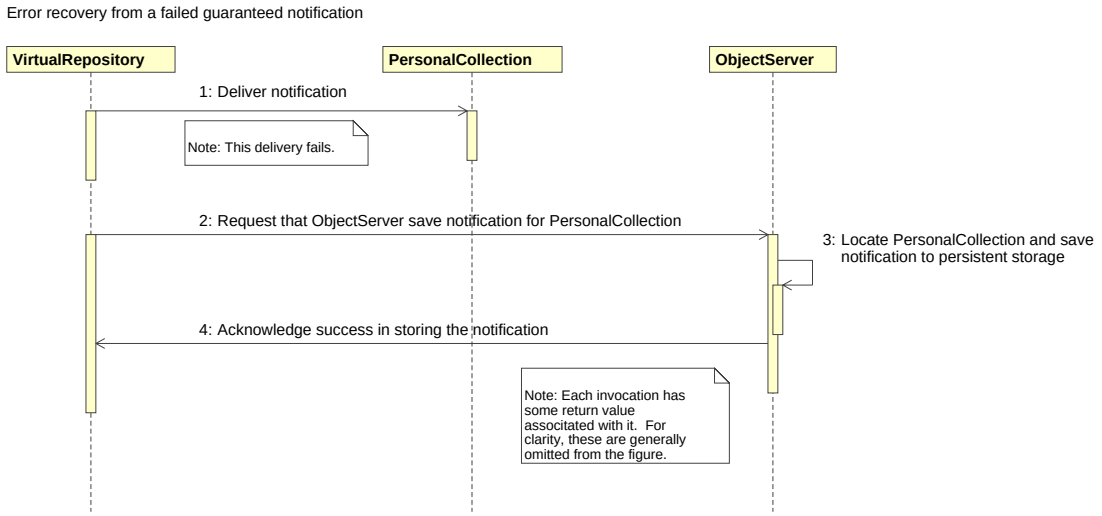

Figure 5.9: Error handling with mailbox notification delivery.

with pending notifications, and the PersonalCollection can handle them as necessary. This error recovery mode provides guaranteed notifications, which are mentioned in terms of a library context in [HF99]. This is similar to the postman leaving a package on your doorstep if you are not home; eventually, you will arrive and pick up the package and the postman (VIRP, in this case) has performed due diligence in delivering it. This implements a form of guaranteed notification delivery as described in Section 3.4.2.

The second and more advanced error recovery mode, shown in Figure 5.10, occurs when the VIRP contacts the ObjectServer of the PersonalCollection and the ObjectServer restores the PersonalCollection object from persistent storage for notification delivery in step 4. In this case, the PersonalCollection receives, in step 6, the notification in near real time relative to when the VIRP sent it. This model is equivalent to the postman having the recipient sign for a piece of mail to guarantee its delivery; an unsuccessful delivery will raise an exception in the VIRP propagator. As described in Section 3.4.3, outside of the event of an error preventing notification delivery the guaranteed and guaranteed immediate notification delivery methods differ only when one of these error routines must be executed. In

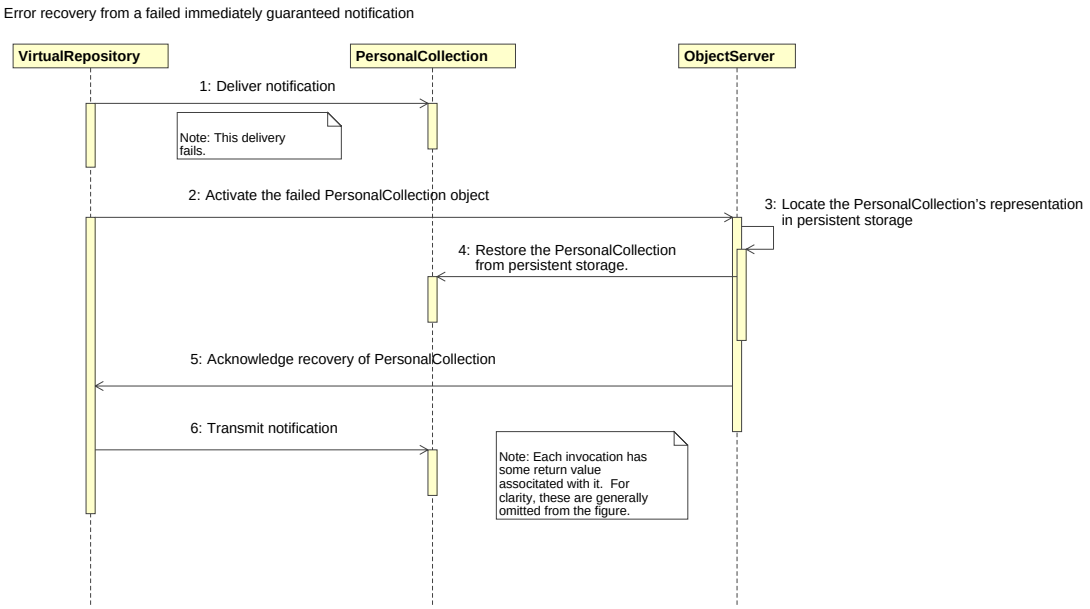

Figure 5.10: Error handling with in-hand notification delivery.

this last case, Hinze and Faensen [HF99] describe something similar called real-time notifications in which they require additional infrastructure support used to restart failed system components. This infrastructure for error recovery here is provided by the persistence layer and ObjectServer in the PIE system. This implements a form of guaranteed immediate notification delivery as described in Section 3.4.3.

Network, ObjectServer, and / or VIRP failures are not covered in this model because they fall outside of the criteria presented in defining the meaning of the term "guarantee." In the event of such failures, however, the system relies on the eventual semantics of the notification model for error recovery. The PersonalCollection objects will eventually receive notifications of changes from the VIRPs about the information providers for which the VIRPs are responsible. In addition, if the PersonalCollection objects are not functioning for some reason, as soon as they are able, the PersonalCollection objects have the ability to query the VIRP, and this could be implemented as another error recovery technique.

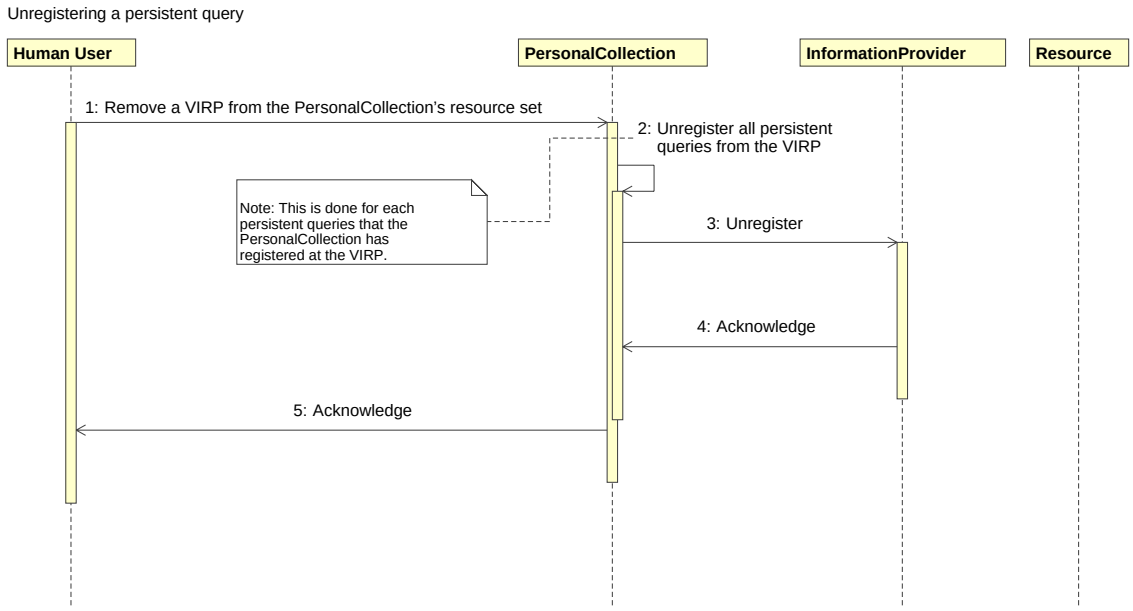

Figure 5.11: Unregistering a persistent query in PIE.

This is a policy decision that can be implemented in the PersonalCollection, but there is nothing in the protocol that limits the PersonalCollection from acting in due diligence for the user in recovering from errors. In the case that the PersonalCollection were to take such an action, this sequence would simply begin with step 4 in Figure 5.8.

For the sake of completeness, we show the process of unsubscribing from a VIRP in the event that a user removes the VIRP from the user's PersonalCollection information provider set; this appears in Figure 5.11.

5.7 Networks of PIE Servers

Thus far, the discussion of PIE has focused on a single instance of a PIE server. Clearly, in order to have a useful persistent querying system, many PIE servers must exist throughout the network in order to support different users and different information providers. The distributed case follows from the single PIE server because the naming conventions

described and used by a local server can be used to locate and interact with PIE objects throughout a wide-area network. The ObjectHandle format convention:

$$< \text{server IP address} : \text{server port number} : \text{context name} >$$

scales to address the entire internet. The missing piece is how widely spread PIE servers perform distributed object discovery, the result of which is an address of the form above. This function can be performed by a PIE-level name service that, minimally, has registrations for all VirtualRepository and shared PersonalCollection objects participating in PIE. Once ObjectHandles have been served to a requesting client, the local PIE server can resolve to the remote PIE objects using the handles as easily as contacting a local object. This name service is described in the scalability discussion of PIE in Chapter 6.

5.8 Conclusion

This concludes the implementation of persistent querying in the PIE system. Clearly, there are many additional avenues that are available in terms of options for error recovery and querying VirtualRepositories. A stable and complete framework has been developed that implements the persistent query protocol in full and, in cases such as searching an information provider using timestamps, in the worst possible scenario. The implementation was successful in proving that it is reasonable to place requirements on information providers, create wrappers for information providers, and make the guarantees we make in the persistent query protocol in the PIE system without employing significant machinery to support the guarantees in the VIRPs or at the information providers themselves. The next chapter analyzes the persistent query protocol in PIE in terms of the requirements set forth at the beginning of this thesis, especially scalability, efficiency, and requirements placed on the information providers.

6

Analysis

6.1 Overview

Thus far, we have discussed the implementation of a PIE server in a single server environment. One of the goals of PIE and persistent querying is to create a system that will operate on internet scale. This requires a high degree of flexibility in the system and independence for every participant to meet the resource needs of their own locale. In addition, our requirements have stipulated that persistent querying be a lightweight addition to an information retrieval system and that the requirements placed on an information provider's participation in the PIE persistent query system be reasonable. This chapter will analyze these criteria after looking at the flexibility of persistent querying in PIE.

6.2 Flexibility of Design

Working on internet scale also requires working on internet time. While a hackneyed phrase, the implications of internet time are significant in terms of the speed at which an information retrieval system on the internet must evolve to meet the demands of users and to keep them informed. Paramount to meeting this goal is system flexibility and evolvability. This is why PIE and persistent querying are such a good match. The object model of PIE, and especially the encapsulation of information providers in VirtualRepository subclasses

implementing a common VIRP interface, provides implementors of new VIRP types the ability to work in isolation of all other components of the system. The time to create a VIRP is as short as the time necessary to provide ways to determine its metadata, wrap its search interface, and parse its search results. If an information provider does not provide features such as timestamps, more effort will be involved, but we anticipate being able to accommodate these issues in PIE. As shown in Figure 5.5, the VIRP inheritance hierarchy hides the functionality that deals with user proxies and notifications in the base class. Thus, subclasses are free to concentrate on implementing the important functionality for a given information provider. In addition, the coupling degree of persistent querying to the objects involved, PersonalCollections and VirtualRepositories, is very low. The persistent querying functionality is concentrated in these two locations and can be evolved independently of other components in the system. In all, the PIE object model and lightweight design of the persistent query protocol yields a framework that is flexible enough to be adapted to user needs, especially when trying to satisfy users' insatiable desire for more information providers. Implementation of persistent query functionality could be accomplished in information systems using these same design principles.

6.3 Scalability

Of the utmost importance in the PIE persistent query system is facilitating users' meeting their information needs. Scalability of the persistent query system and PIE is the next most important issue of concern for implementing this SDI system on internet scale. We believe that the PIE platform will scale very well in a system having tens of thousands to millions of objects. Building PIE on top of CORBA is a significant contributor in this optimism. CORBA is designed by nature to be a highly scalable, distributed, and object-oriented infrastructure. Scalability is provided by the ORB on top of which PIE is implemented. Sites maintaining PIE servers can choose an ORB that matches their economic constraints and that scales to meet the resource demands placed on their server infrastructure. This is

possible because of the interoperability of CORBA ORBs implemented by different vendors. While we are not using this flexibility currently, we allow individual PIE servers to execute on top of a platform providing functionality necessary to meet user demand and scalability needs while at the same time permitting heterogeneous CORBA ORBs to work together. In principle, this is possible, but in practice, this has been difficult to guarantee. Recently, however, the interoperability of CORBA ORBs is improving as the OMG closes holes in the CORBA specification that caused interoperability problems between ORBs from different vendors. CORBA scalability and response time has been the subject of significant research in recent years. Schmidt [GS97, GS98, POS+99] is the foremost expert on such issues and has broadly studied existing CORBA ORBs for their real-time and scalability properties. In early work, Schmidt *et al.* found that CORBA implementations incurred high latency and high method invocation cost because of layers of object-oriented abstraction present in CORBA; low scalability resulted partially from these issues [GS97]. Later research revealed that it is possible to overcome these impediments and create efficient CORBA ORBs [GS98]. As the CORBA specification, hardware capabilities, and ORB development has evolved, CORBA ORBs have become more heavily multi-threaded and capable of handling a large number of objects at an ORB and clients invoking methods on those objects [POS+99]. Schmidt's research has culminated in a real-time CORBA ORB that is available for free on the internet[1]. While this ORB is implemented in C++, the optimization techniques applied to the CORBA platform are portable to other ORB implementations. The CORBA platform is scaling more effectively as it matures, and while CORBA is based on the principle of abstraction, which always incurs some overhead, we believe that our distributed architecture, that does not concentrate large numbers of objects at any single PIE server, will scale well on top of CORBA. Thus, by distributing the load on PeC and VIRP objects spread throughout the system and building on an inherently scalable platform, the persistent query functionality and PIE architecture should scale well.

Prior discussion of the PIE implementation has mentioned the requirement to have a

[1] `http://www.cs.wustl.edu/~schmidt/TAO.html`

naming authority that services name requests and is knowledgeable about the objects existing at distributed PIE servers. In fact, such a name server is fundamental to an effective distributed PIE system. Obviously, having a centralized naming authority is a dangerous prospect, especially in a system that relies on discovering information providers. We propose that a solution to this problem be solved through the use of an inexpensive and highly scalable web server such as Apache[2]. The name server will receive periodic updates from PIE servers with new VirtualRepositories that have come on line and PersonalCollections that are shared by their owning users. These updates consist of metadata about each object and the object's ObjectHandle. This will give the name server the ability to specify an object of interest by name and provide some metadata to describe the object. The name server will receive queries about other objects, which may be executed over the indexed metadata of each, and will return lists of ObjectHandle objects in response. Users will issue such requests and the ObjectHandle list will be used to resolve to specific objects returned by the name server. We believe that this solution will provide a highly scalable solution to the issue of providing a distributed name service for PIE. HTTP servers are a good match for this task because of their bias toward handling short transactions that may involve databases or information retrieval engines, and HTTP servers can scale to handle millions of transactions. Operational examples of such scalability exist at many websites such as Google[3] which handles millions of transactions each day. Significant research and practice has been invested in creating scalable and reliable web servers and platforms. We will rely on these successes and believe that they will provide a sound solution to scalable information provider discovery and persistent querying in PIE.

In our current implementation, we have tested using a single PIE server that has run under a load of multiple users, each with multiple PersonalCollections, and up to 100 VirtualReposotory objects, each representing an NCSTRL information provider. This has been done on a single processor machine with 256 megabytes of main memory and a single network interface. The memory footprint of such a system is fifty-five megabytes. One of

[2] `http://www.apache.org`
[3] `http://www.google.com`

the reasons that we are optimistic about the scalability of PIE and persistent querying is that the maximum amount of traffic generated by each user will be limited to how quickly users can process the data they are presented. Construction of a PersonalCollection is performed entirely on the client-side, and messages are sent to the server only as often as necessary. The use of variable rate change notifications reduces the amount of network traffic and processing between the PersonalCollection and VirtualRepository objects. A persistent query system is not as casual as surfing the web; an investment must be made up front by each user to tailor a PersonalCollection to their information needs and to then pose meaningful persistent queries over the information providers.

Because of the distributed architecture of the PIE system, the load on each individual server will be reasonable, with the majority of interaction occurring between the user interface client and the PIE server on which the user's UserProfile object exists. Processing a persistent query at a VirtualRepository simply requires issuing the query at the information provider the VIRP wraps. On the internet, this would be done through a CGI or similar search interface; these scale to handle the required traffic at many internet sites. At other information providers such as digital libraries or databases, their user bases are likely to be different than the web at large. For example, the Virgo[4] search interface at the University of Virginia has a different size of information consumer audience than does that at `http://www.cnn.com`. Information providers that are under heavy demand will likely need to be hosted on PIE servers and networks that can handle that demand. This issue is also one of using a scalable CORBA ORB and is in principle no different from handling many page hits at an active website such as that for the Olympics[5]. All other operations, such as PersonalCollection construction and browsing search result lists, take place at the leisure of the system or users and are not required to provide "real-time" results that place demands on third parties.

In terms of storage requirements, persistent queries place a storage demand in several locations. Foremost, persistent queries will always have to store result lists on a PIE server

[4] `http://virgo.lib.virginia.edu`
[5] `http://www.olympics.com`

at which a user's PersonalCollection objects are hosted. In addition, the persistent user-created search result folders will have to be stored in a similar location. This storage will be cumulatively significant over the entire PIE system, but by using distributed PIE servers, this requirement is amortized over all of the PIE servers. Administrators at each of the servers will simply need to assure that there is enough disk space to accommodate the local user-base. At extremely large PIE servers, the size of the user-base may dictate that an alternate to disk based storage be used and a database may be used instead to speed storage, speed retrieval, ease synchronization, and increase the scalability of storing user folders and search results. In practice, such a solution would be easy implement because of the FolderManager interface presented in Figure 5.4 and would scale well based on experience with database use at large internet sites such as search engines. Another location in the persistent query system where database infrastructure will be useful is if an information provider does not provide timestamps. In this case, it would be very effective to leverage the power of a database to filter and store the document identifiers that have been previously returned to the user by a persistent query, for each persistent query the user has posed. Currently, timestamps over the NCSTRL-based information provider test-bed are implemented by differencing a flat file on disk with a current search result to yield the complement of their intersection that is not already stored; these are the new search results to be returned to the user. This could easily be performed with a simple SQL query over a database storing the same document pointers.

In each of these areas, distributed object discovery, distributed object naming, individual PIE servers, computation, and storage requirements, the persistent query implementation in PIE should scale well. For those areas that will have an issue with the current implementation, a solution has been presented to be tested and used when a suitably large test-bed can be constructed.

6.4 Efficiency of Persistent Querying

Outside of considering the scalability of the system, the efficiency of persistent querying is of importance to this thesis. The resource requirements for persistent querying are the following. First, the VirtualRepository objects must keep track of a set of

$$< \textit{subscriber\_id}, \textit{query\_id}, \textit{notifications\_enabled}, \textit{notification\_pending} >$$

structures for each subscriber and registered persistent query. Assuming the size of this structure is on average 200 bytes, a table representing one million queries would total 200 megabytes. This may seem large, but in terms of the storage capable in a modern database, this storage is quite small and could be handled easily. In the case of scaling persistent querying to this magnitude, databases, as discussed earlier, will be necessary throughout the PIE system which has been designed with this eventuality in mind. In addition, the VirtualRepository object will become a bottleneck when having to notify all of one million queries when a change notification is received from the information provider. For this technique, several optimizations can be made. First, only the $< \textit{subscriber\_id}, \textit{query\_id} >$ identifier would be kept in the four-tuple above. The PersonalCollection subscribing to the notification would have to handle de-multiplexing the single notifications onto the potentially many persistent queries that may be posed over the information provider. Second, clustering of notifications may be necessary and could be performed at the granularity of individual PIE servers. This would simply require a layer between the PersonalCollection objects at a server and the larger PIE federation as a whole. In this case, the subscription of a PersonalCollection to a VirtualRepository will not occur if the PIE server hosting the PersonalCollection is already subscribed to hear the VIRP's change notifications, and the largest number of subscribers at any one VIRP will be the total number of PIE servers that are present in the entire federation. The final cause for concern in scaling a persistent query at a VIRP is handling the many queries from the million subscribers that may arrive on the VIRP. Modern commercial CORBA systems have been shown to perform well as transaction processing platforms. A query is a simple transaction between a PersonalCollection

and a VirtualRepository, and it can be treated as such by using thread pools, load balancing, and other techniques on the VirtualRepository to limit the load placed on specific objects. These techniques are well studied and could be implemented here either natively in PIE or by using a transaction platform available in commercial CORBA systems. Schmidt [Sch01] conducted a study of such threading techniques in CORBA serves.

Handling persistent queries at PersonalCollection objects is a much easier task in terms of the resource requirements placed on the infrastructure. Most importantly, the PIE model is inherently distributed, so individual PIE servers will be able to process their own objects' requests. PersonalCollection objects are mostly concerned with keeping track of current subscriptions and issuing queries to VirtualRepositories. This process, considering that PIE has already been tested with over one hundred fifty CORBA objects running on a single machine, is entirely tractable considering that we expect only hundreds of objects to be hosted at a single PIE server.

In terms of network bandwidth, several areas deserve consideration. First, subscriptions from PersonalCollection to VirtualRepository require several hundred bytes of synchronous method invocation to complete. The same is true of unsubscriptions and change notifications. The significant overhead is in issuing and processing queries at VIRPs. In this case, the PersonalCollection must send its ObjectHandle identifier and persistent query to the VIRP. The VIRP must then send the query to the search interface of the information provider. If the provider supports timestamps, the returned search result will generally be much smaller than if the search interface does not support timestamps. This is because the time stamped search result will not contain older search results that the latter would contain repetitively. The network requirement could be significant in terms of having to return these results to the user. There is no way to avoid performing this action; it is the purpose of an SDI / persistent query system and is unavoidable. Returning document pointers, as done in the current test-bed, instead of returning actual documents should reduce this load. We do not expect full documents be returned to users in a search result and find caching to be a more scalable and practical approach to this problem, see Section 3.4.1.

In addition, current calls to execute a query are synchronous, and leveraging features of a CORBA implementation to perform asynchronous calls may increase fault tolerance and efficiency of querying.

Given this characterization of the resource requirements of a persistent query, we believe that the protocol presented here will scale well and will not present any significant burden to network and server resources that has not already been mentioned. Several optimizations have been provided to address significant concerns.

6.5 Requirements on Information Providers

In order for any SDI system to be successful, the information providers that are at the disposal of the users are extremely important, and their participation is essential to providing a useful service. As a result, the cost of participation by such providers must be kept low, and this is why one of our requirements has been to keep the requirements placed on the providers as minimal as possible without overburdening them. Our only requirement for information provider participation is that they emit a change notification to a single subscribed object, the information provider's proxy. The use of a proxy in this location is important in reducing this burden because without a provider proxy the number of user proxies subscribed at the information provider is virtually unlimited. Use of a provider proxy as the SDI system's interface to the information provider keeps the cardinality of this subscriber set on the provider to size one per information provider. This absolves the information provider from having to provide a subscription interface, to provide an interface to turn off notifications, and to maintain all of the state associated with handling subscriptions and propagating notifications to many subscribed user proxies. Notifications as the only requirement on providers is a reasonable expectation for participation in an SDI system.

A search interface and timestamps can be emulated in the infrastructure of the information provider proxy, though at great cost. We believe that at least the former will be

present in virtually all information providers; in the internet age, provision of an interface to search content is almost a de facto standard at information sources. Timestamps are more difficult to require, but any monotonically increasing identification number will suffice. As SDI systems supporting persistent querying and a diverse information provider set become more prevalent, the economic models associated with participating in such systems are likely to motivate information providers to provide timestamps. We can foresee some of these services subscription based, even simply at the information provider level, in the future.

This requirement on an information provider is a reasonable, scalable, and simple to provide feature that allows an easy entry of information providers into a persistent query infrastructure similar to that implemented in PIE. The complex part, wrapping the functionality of the information provider, is the responsibility of the implementor of the Virtual-Repository wrapper, a barrier reduced from the information provider's cost of participation in the persistent query system by effective abstraction.

6.6 Conclusion

The distributed, notification-based architecture of the persistent query system in PIE are significant factors in the scalability of the system. These characteristics facilitate the inclusion of significant numbers of information providers and user proxies spread throughout the internet on distributed PIE servers. Use of an HTTP based centralized name server could serve millions of object references daily, and the use of databases reduces access latency and increases the usability of stored data. The architecture reduces the processing costs of information provider proxies and user proxies by spreading them out on many different machines. This architecture and the solutions to potential bottlenecks should lead to a system that will facilitate users' meeting their information needs through the use of the persistent query functionality present in the PIE system.

7

Related Work

7.1 Overview

This chapter discusses work done by others that relates to research described in this thesis. Research presented here is decomposed topically and temporally. The work is decomposed into the following topics - the history of SDI, active databases, event notifications systems, SDI research in the 1990's, and continual queries.

7.2 SDI in History

SDI is one of the older application areas in computer science. Luhn's seminal paper [Luh58] on the subject was presented in the second volume of the IBM Systems Journal. Luhn described a system that automatically abstracted and encoded documents so that they could be matched to machine-learned user interest profiles. Luhn believed that information retrieval functionality, then a young field, was necessary to make such a system a success. A document is auto-indexed by determining the frequency of occurrence of significant terms in the document. User profiles, called action points, are created similarly to the representation of documents. Matching is performed when new documents arrive in the system and those documents that "sufficiently" match user profiles are printed and the document provided to the user. Interestingly, Luhn required human intervention to transcribe text documents

into digital format and to formulate user profiles, tasks which would be automated or performed by the user today. The goals of the system Luhn describes are still the goals of a modern SDI system, which we describe in terms of persistent querying. In many ways, Luhn's ideas were ahead of their time in SDI and computer science, especially his ideas of the machine learning of user profiles. Housman [Hou73] provides an excellent survey of the systems implemented based on Luhn's ideas during the 1960's and early-1970's. Mainly used in libraries, SDI systems of the time allowed users to create profiles that were stored in a computer system. Profiles could consist of Boolean phrases or other queries that were evaluated against newly arriving documents. The results of such queries were returned to the user in the form of photocopied journal papers. SDI systems tied together several databases, including information items such as bibliographic references, so users could pose one query that is transmitted to all of them. One of the concerns of supporting working SDI systems was the cost of buying computer time on mainframes.

While the goals of SDI systems have been understood since Luhn's 1958 paper, recent work has focused on making SDI systems scalable and capable of operating over heterogeneous information providers. Yan and Garcia-Molina [YGM94] make the point that in the past, SDI systems were centralized, but in order to scale up to work on internet scale, SDI systems must be distributed. In this paper, they consider the efficiency of such a wide-scale system. Their focus is different from ours, though, and rests on "distributed matchmaking." They describe a system where distributed servers accept user profiles and documents from information providers, match them, and distribute documents to users. This is in lieu of either distributing all documents to all profiles or putting all profiles at all document servers. Altinel and Franklin *et al.*[AF00] describe filtering XML documents for SDI using an XML specific query language and a query engine for executing such queries over the structured XML documents.

7.3 Active Databases

One means for implementing a change notification service in a database environment is with an active database. Active databases are concerned with observing changes and triggering events based on the changes in a database. These systems are based around the event-condition-action (ECA) model where the database monitors a query and reacts, performing an action, under certain conditions observed once the database is changed. Patton and Díaz [PD99] present an excellent survey paper of the field. Their survey addresses paradigms for database triggers and discusses specific systems that implement such functionality. McCarthy and Dayal's [MD89] work details the architecture of such systems. Active databases are well studied and understood and are not very well suited to the persistent query environment because of scalability issues, though triggers have been used to implement continuous query functionality as will be described shortly. Also, internet-based information providers are not in the practice of providing an interface to facilitate access to their trigger mechanism on databases storing their content and prefer to provide such information through an information retrieval based search interface.

7.4 Event Notification Systems

The effectiveness of an SDI system hinges on its ability to either tell users about new documents or distribute them directly to users. In some ways, both models rely on the propagation of events within a system that supports matching user profiles with information items (or vice versa). While we do not use a specific event notification system, implementing PIE or persistent querying on top of such a foundation would not be difficult. Significant work has been done in the field of distributed event notifications. Wolf and Rosenblum [RW97] present a model for internet-scale event notifications. They evaluate previous work in event notification and describe the shortcomings of other systems. Then, they propose a framework consisting of seven models, the levels of which address different aspects of event observation and propagation. The models are object, event, naming, observation,

time, notification, and resource. Each level is used to describe and to facilitate some aspect of distributed event notification such as subscription, the description or detection of relationships between notifications and events, and a way to name components of the system. The authors consider scalability issues in the naming of, propagation of, and detection of events. They also describe pattern detection and trends over time where we are simply concerned with the occurrence of the event and leave pattern modeling up to the event consumer. The dimensions presented, however, are a complete list of those necessary in any event notification system and include many of the steps that we implement in persistent querying such as expressing interest in an event, observation of events, notification of events, distributed object naming, and response to events. Many of the aspects of the high-level framework they present are fundamental to constructing any successful distributed, persistent query system. Hinze and Faensen [HF99] present a model for an internet scale alerting service, describe the shortcomings of that presented by Rosenblum and Wolf [RW97], and diagram a system architecture. Hinze and Faensen describe another model of event dissemination usable with or without polling "suppliers", our information providers, participating in the system. The architecture is similar to that in PIE where suppliers interact with an "alerting service" that stores query profiles and has a notification buffer. These two components are linked with an event observer that can reside in either the supplier, similar to our requirement on information providers, or in the "alerting service" component. The authors also mention some of the guarantees that can be made to clients in such a system including guaranteed and real-time delivery, our guaranteed and guaranteed immediate notifications respectively. The alerting service was realized in the MediAS system which facilitates change notification for content in a digital library; observers are used as wrappers for suppliers' heterogeneous implementations and normalize their interfaces to the MediAS system as VIRPs do in PIE. The paper and MediAS work, however, does not consider the scalability of the system or issues related to lossy suppliers.

Several researchers have investigated frameworks and dimensions for supporting dis-

semination based systems. Crespo and Garcia-Molina [CGM97] describe the spectrum of awareness services in digital libraries along several dimensions, push or pull of notifications, stateful and stateless clients and servers, the cardinality of clients at a data store, and the awareness level of stores to sources and vice versa. Franklin and Zdonik [FZ97] describe similar dimensions including the push or pull of messages at a periodic or aperiodic rate in a unicast or multicast environment; these aspects are described in a model called DBIS. Franklin *et al.* [AAB+98] expand this model in later work to describe the design options for "nodes" that are used to compose different data distribution models; these include classifying the data source based on the modifications it makes to an information stream (no modifications, some modifications, etc.), the caching model used, optimization of the push schedule, recovery services of nodes, and functionality of value-added nodes that might provide such services as merging. It is noteworthy that the caching notion presented by Franklin *et al.* is used to optimize page fetching for speed, not to preserve volatile information items. Franklin *et al.* [AAB+99] have since realized the DBIS toolkit. The characteristics described in these papers are useful in considering our persistent query functionality, and all are broad enough to contain the characteristics of our implemented persistent query system in PIE. Conceptually, the dimensions we describe for supporting persistent querying are similar to these bodies of research, especially Franklin's work [AAB+98], but with a different perspective. Crespo and Garcia-Molina [CGM97] describe a "hint-pull" mechanism as an alternative to purely push or purely pull mechanism where the information provider hints to interested clients that a change has occurred on the provider. Our support dimensions are concerned with describing practical means for implementing persistent queries as opposed to classifying a system. Allowing for a hint-pull client / server interaction model, our persistent query protocol fits into any of these awareness or dissemination frameworks.

7.5 SDI Research in the 1990's

Several important SDI systems were constructed in the early 1990's. Pasadena, Tapestry, and SIFT used similar information streams, including Netnews, to create systems that distributed information based on user profiles. The techniques for querying information providers to access information items ranged from SQL queries over databases to information retrieval techniques.

Wyle and Frei [WF89] describe Pasadena, a wide-area SDI system. Pasadena creates a selective dissemination of Netnews service that uses an information retrieval based query system instead of a SQL based database system. Pasadena provides indexing, querying, dissemination, archiving, extraction and selection functionality in a single environment with the services available over wide-area networks. The system runs over several heterogeneous and distributed information providers using a WAN information server which receives streams from each provider and which has components to index and archive Netnews and other information items. The Pasadena SDI service, under development at the publication time of the paper, provided user profiling functionality to match incoming information items to interested users. Pasadena is particularly concerned with the scheduling of queries that are executed at remote information provider sites. Pasadena attempts to consider the update rate of information at remote providers and the network failure rate between the client and information provider in order to optimize a query schedule to balance query transmission success with the chance of finding matching documents. A sliding window is used to maintain a weighted knowledge of the most recent attempts to execute a query at an information provider. Wyle and Frei later describe the Pasadena system [FW91] implemented with SDI functionality and user profiling. This later work describes the results of using two different indexing systems in Pasadena. Every user has one user profile that contains one or more queries. Queries can list the information providers to search, search patterns to include / exclude, and a time interval for the query rate. User queries are run periodically by polling the information providers. This work is similar to our persistent

query work in that it attempts to provide a reliable and distributed system for disseminating information from heterogeneous information providers. Wyle and Frei describe issues with such providers well, terming the differences in query format and interaction with each provider as "highly-parameterized," a phrase that accurately describes the problems that PIE's VIRP component addresses.

Terry *et al.* [TGNO92] introduce continuous queries in an append-only database environment. They describe the difficulties of developing a continuous query mechanism using the naive solution of simply executing a query repetitively and note that such a methodology may create nondeterministic results, duplicates, and inefficiency. They present continuous semantics as a solution to the problem; continuous semantics say that the results of a continuous query are "the set of data that would be returned if the query were executed at every instant in time." The theoretical framework implementing these semantics requires append-only databases and timestamps for every item. Queries run against the database are only run against the "new" items in the database. Queries are SQL based and must be monotone (have a temporally, non-decreasing result set) so as not to have difficulties in partitioning the database as the state of the database state changes. Terry *et al.* had to take care in implementing continuous queries over Netnews articles because as readers respond to different threads, the past content of the database changes; they addressed this through the construction of the tables in the database. Queries are executed periodically by polling the database. Terry *et al.* describe Tapestry [GNOT92] which implements their continual query semantics in an e-mail, Netnews, newswire environment. Information items from these information streams are extracted, indexed, and inserted into a database. Users pose continual queries to the Tapestry system which are executed occasionally against the database to discover any new information items since the last query execution. The results are e-mailed to the user; documents in the database store are persistent. Persistent querying is similar to this body of work in concept but very different in practice. The append-only information environment is somewhat unrealistic when considering internet based information providers. In addition, users can not be expected to always pose mono-

tone queries to information sources. The persistent query protocol described in this thesis is also scalable and accounts for information providers that do not export a SQL database. We attempt to emulate the continuous semantics on the internet in a reasonable fashion considering the characteristics of this environment but without overly constraining the information providers with an append-only requirement. Given an append-only information provider with timestamps in our persistent query system and relaxing the monotone query requirement, Terry's strict continual semantics will be met with our framework.

A more recent information filtering system described by Yan and Garcia-Molina [YGM95] is the Stanford Information Filtering Tool (SIFT). The toolkit provides an e-mail interface at which users can pose profiles they have constructed and tested over a test collection. At a periodic interval provided by the user, SIFT evaluates the user's profile against Netnews articles that have accumulated in the SIFT server's document collection. The architecture of the system provides a centralized server at which Netnews articles accumulate and at which users pose profiles. SIFT uses the WAIS [KM91] tools to index and search the news articles. While real information retrieval techniques (as opposed to SQL queries and databases) are used to search documents that have been indexed, the original SIFT system was centralized and only considered the Netnews format as opposed to providing for arbitrary information providers. More recently, Yan and Garcia-Molina [YGM99] discuss a distributed version of SIFT. In a distributed configuration, SIFT uses quorums to decide how to distribute documents and profiles in the network of SIFT servers. A single document in the system would be sent to a a set of SIFT servers called a document quorum and likewise for a profile and a profile quorum. A guarantee about the opportunity of a profile to see a document can be made if the intersection of these two quorums (document and profile) is non-empty for all documents and profiles. The difficulty with this configuration is that it requires cooperation of the information providers or complete knowledge of the information items as in a Netnews environment. This would be very difficult to implement over arbitrary web information providers for this reason and also because of issues in scaling SIFT to millions of information items.

7.6 Continual Queries

A significant body of research has been performed by Ling Liu, Carlton Pu *et al.* previously at the Oregon Graduate Institute and currently at Georgia Tech. The research conducted at these locations has focused on what they term "continual queries" and on a framework for query processing. Liu and Pu [LP97] address the integration of distributed, heterogeneous information sources (databases) in a query framework that provides services to process queries; the system is realized in the Distributed Interoperable Object Model (DIOM). DIOM describes information provider wrappers, similar to VIRPs in PIE, and a mediation layer between clients and information providers. In this middle layer, query decomposition, query routing, query scheduling, and results merging are performed for user queries. The issue with these features is that they are described in terms of "relevant" and "optimal" decisions made for routing queries to "relevant" information providers; from an information retrieval perspective, these terms are meaningless and no exposition on the algorithms used or justification for these claims is made. In the database environment over which DIOM executes, such claims can be self-fulfilling because the representation of an information provider can be precise if properly constructed from the database; this is not the case in an information retrieval setting where creating exact summaries of arbitrary information providers is much more difficult. DIOM is used as the foundation for other continual query work performed by Liu and Pu.

Liu *et al.* [LPBZ96] describe continual queries as standing queries that monitor information providers of interest and provide matching results to users. They present a differential re-evaluation algorithm (DRA) that is used to incrementally update the previous results of users' queries instead of having to re-evaluate a query each time it is posed to a set of information providers. They describe the DRA algorithm relative to Terry *et al.*'s [TGNO92] continuous query work. DRA is accomplished by differencing the results of the current query with the previous query for every query at every database; only the difference of the two (the new part) is returned as a query result. A sliding window is used to discard old

results. Liu *et al.* provide an algorithm to accomplish this in terms of databases but do not address issues Terry *et al.* do such as monotonic queries. DRA is similar to the worst case situation described in Section 3.3.3 where the user proxy must difference consecutive search results to ensure that users receive only the newest results except that DRA performs the computation at the database instead of distributed on user proxies. The former solution incurs a scalability problem. As with DIOM, a significant difference in this work from ours is that of the database environment versus an information retrieval one. The former environment is very precise in the presence or absence of tuples that match a SQL query; information retrieval is concerned with identifying information items that are relevant to users' queries while allowing flexibility in expression of interest and latitude in determining a document's relevance to a query.

Pu and Liu [PL98] expand on their continual query work and present a software architecture for implementing a continual query system outside of DIOM. This architecture consists of three key components, the event driven update monitor, the trigger firing daemon, and the continual query evaluator. In this architecture, most data is delivered to users by a server push method where the server evaluates new information items against users' queries stored on the server. The architecture expects to be able to parse users' queries for a termination criterion and expects a trigger that can be turned into a SQL query to evaluate at a database. As in DIOM, Pu and Liu use the term "relevant" to describe selection of information providers. The architecture, while presented in the context of servicing the internet, is not discussed in terms of scalability.

The DRA and continual query work is realized in a three implementations, OpenCQ[1] [LPT99a], JCQ [LPT99b], and WebCQ[2] [LPT00]. OpenCQ [LPT99a] is an implementation of Liu and Pu's continual query framework that provides users the ability to register queries that will be executed continually under the equivalent of the DRA model. In this work, Liu *et al.* have two event observation methods, synchronous using database triggers and polling and have dropped content based changes from the OpenCQ implementation.

[1] `http://www.cc.gatech.edu/projects/disl/CQ/`
[2] `http://www.cc.gatech.edu/projects/disl/WebCQ/`

Polling is relied upon to detect changes in snapshots taken over consecutive time intervals at information providers. After detecting a difference between the representations, a change notification is fired and the OpenCQ system decides which continual queries must be executed, schedules them, and executes them. Liu *et al.* claim that their model outperforms polling over a pull-based delivery model for a large number of objects but make no comments about handling a large number of queries, a potential problem in such systems. This claim is easy to support when polling information providers at a thirty second interval for all subscribed clients; the polling and query execution scheduling for all clients is done by a single entity yet the system is still polling, which is a limitation to implementing such a system in a large scale environment. JCQ is a Java implementation for monitoring web information and provides a scalability enhancement over OpenCQ.

JCQ [LPT99b] attempts to cluster registered continual queries into groups based on similarities in the triggering condition of the query. By doing this, the trigger need only be checked once for the group as opposed to once per query as in OpenCQ for every change at an information provider. Liu *et al.*'s assumption facilitating this is that triggering conditions are all similar except for "the appearance of different constant values in the trigger specification." The architecture for JCQ is the same as the conceptual system and OpenCQ, and determination of change is still done by polling the information provider. The same issues with OpenCQ and DIOM exist with this architecture with the added problem of assuming that the triggering condition for a continual query can be classified and that large numbers triggers differ only by a constant. The example queries provided are of a form that detects a percentage change in the price of a stock; these may be simpler to classify than free-text queries posed to an information retrieval system, but they only meet a limited set of users' information needs and constructing a system around such an expectation is unrealistic. In addition, in discussing the scalability of continual queries, only a maximum of twenty-five are considered and no discussion is presented for the scalability of DIOM on top of which JCQ is implemented.

WebCQ [LPT00] is a web information monitoring system that is somewhat different

from the other continuous query systems discussed. The system provides functionality whereby users may express interest in the structure or characteristics of arbitrary web pages, forming a web-based continuous query. WebCQ is then responsible for detecting changes between two versions of web pages in terms of a number of "sentinels" such as a page's byte count, content, or HTML structure. WebCQ attempts to exploit the semi-structured nature of HTML to detect changes in image, link, table, and other HTML specific structures. The WebCQ infrastructure caches pages from web sites for speeding subsequent requests for the same page, and an HTML differencing engine is used to detect changes between two pages. The differences discovered in a new page are presented to the user showing the old and new page side-by-side, only the differences, or old and new merged together. The fundamental assumption of this system, however, is that for the lifetime of the query a web page's structure does not change often enough to interrupt users' posed WebCQs; such a change would make completing users' queries impossible because the structure of the page over which a regular expression is posed will no longer match. In practice, this assumption is tenuous at best as the structure and / or content of web pages may change often while the page's appearance remains the same. In addition, caching pages internally will yield different or stale search results for different users depending on when the page is updated; the scalability of WebCQ is not discussed in terms of either the number of users, number of queries, or complexity of storing and differencing an arbitrary number of HTML pages. Finally, WebCQ is simply a change notification service that detects whether or not a regular expression or byte count for a page has changed; there is no information retrieval value to the system, it only detects changes and presents the changes to users, a limitation that still requires users to filter the information themselves.

Chen *et al.* [CDTW00] describe the NiagaraCQ[3] system, the continuous query component of the Niagara project conducted between the University of Wisconsin and the Oregon Graduate Institute. NiagaraCQ attempts to provide a scalable, internet based continuous query system. In order to provide scalability, the authors describe optimizations on user

[3] `http://www.cs.wisc.edu/niagara/Engine.html`

queries. Queries are grouped based on users' interests; the examples given all relate to retrieving stock quotes from XML documents. The authors describe using push and pull models for detecting information provider changes. User queries are sent through a query parser and optimizer. As changes occur, user queries are posed against the data sources that, in the two test information providers, are in an XML format. Components of the NiagaraCQ system are used to provide only new content at an information provider for query processing each time a query is executed. Chen *et al.* expect to support millions of users with this system but provide no evidence that the centralized system that implements the NiagaraCQ system will scale, outside of query grouping. Their similarity-based grouping of queries as a means by which to scale to millions of users is another tenuous assumption relying on a high degree of query correlation and fast, efficient grouping of new queries.

7.7 Conclusion

Significant research has been conducted in fields relating to the persistent query work presented in this thesis, which learns from and builds on some of this related research. The event notification framework papers provide the broad picture of system properties necessary to consider when building a change detection or event notification system. Work more specifically related to awareness services and change detection has mentioned how timestamping at an information provider may be used and generally what change notifications can mean. SDI systems such as Tapestry and Pasadena provide work on formal continuous querying and wide-area information dissemination, respectively, and the body of continuous query work from Liu *et al.* has highlighted many points that need to be addressed in the construction of a persistent query system, including the necessity to build a scalable infrastructure and implement meaningful information retrieval techniques. Our work is different from much of that in the past because it facilitates information retrieval techniques to be used for query processing and because responses to persistent user queries are not pushed from information providers to users (or their proxies) as is the case in

many previous systems. Rather, the user proxies are allowed to query the information provider only as necessary and are not compelled to accept search results the information provider unilaterally pushes to the user. In addition, we have addressed scalability issues in Chapter 6, have worked with accepted distributed computing standards such as CORBA, and provide for a heterogeneous information provider set that is not limited to Netnews, or internet web sites but can include any browsable or searchable information provider, including databases, together in a single framework.

8

Conclusions

The purpose of an SDI system is to keep its users up to date with new information items arriving at information providers that match users' profiles. A persistent query mechanism is a method for implementing such a system. In this thesis, we have characterized the solution space for SDI systems and have provided dimensions for supporting and implementing persistent querying. This functionality can be supported at the information providers and in a federation's infrastructure supporting persistent querying, if such machinery exists. Our protocol selects persistent query dimensions that provide reasonable guarantees to users in an SDI infrastructure without incurring significant cost at the information providers. This functionality is realized in our implementation of persistent querying as a subsystem of PIE.

The fundamental requirement that we make of information providers is that they emit notifications when their content changes. While this may seem intricate and difficult to integrate with an existing system, we believe that this functionality can be easily and securely provided by a self-contained, add-on package which can be attached outside of an information provider's existing infrastructure. When a change occurs, the information provider would simply report the change to a notification daemon. The notification daemon would need to provide a subscription interface to the persistent query infrastructure, but in an infrastructure such as PIE, the information provider would never have a subscriber set larger than cardinality one because the VIRP wrapping the information provider would

be the only direct subscriber to the provider's change notifications. The daemon could be constructed to receive and propagate notifications over an HTTP interface, through RPC, through a CORBA method (as is currently implemented), or via any number of other messaging mediums. We expect that such an implementation would be easy to install, could be secure, and would be usable by many different information providers.

Information retrieval techniques are important to allowing users the freedom to pose unstructured, free-text queries to a heterogeneous set of information providers. Use of databases and SQL style structured querying do not provide the granularity and flexibility in search capability that internet users require. Our system differs from previous SDI systems because it leverages users' understanding of how to search the internet with unstructured queries as opposed to strictly structured ones.

Our persistent query protocol and its implementation are a lightweight, efficient, and scalable solution to the SDI problem. We introduce and define several key concepts in persistent querying including variable rate notifications, eventual semantics, and differential querying. Variable rate notifications minimize the rate of notification transmission to subscribers, varying the rate based on a subscriber's responses to those notifications. Eventual semantics are possible as a result of guaranteeing notifications and ensure that a subscriber will eventually see all of its notifications, and differential querying works over internet-based information providers and couples timestamps and a search interface to reduce the cost of querying such providers. These provide a new perspective for analyzing and creating SDI systems which are focused on the needs of the information consumer in terms of customizability, search functionality, and decision making and allow information providers to participate at the simple cost of change notifications and additional searches executed through their search interfaces. This work and perspective show potential for future implementations of large scale SDI systems.

8.1 Future Work

The persistent query protocol and its realization in PIE leave opportunities for significant future work and research. These fall into three categories, conceptual, information retrieval, and systems work. Conceptually, the protocol should be analyzed for additional dimensions which might be supported at the information provider or in the infrastructure. For example, change logging may be useful either at the information provider or at the infrastructure level. While we believe that we have identified the major elements of support necessary for persistent querying, additional value-added features will certainly arise.

In terms of information retrieval aspects of the persistent query concept, many different locations exist for enhancing persistent querying in PIE. Most significantly, information retrieval techniques can be leveraged throughout the infrastructure to potentially enhance the query results returned to users. More interesting results merging and ranking algorithms should be researched and implemented for use when merging the search results of persistent queries. In addition, greater use of effective selection could be used to increase the quality of search results and reduce the amount of work necessary to provide search results to users. The demands placed on search interfaces available at average web sites, not just those sites solely devoted to searching, should be studied and understood. This needs to be done in terms of the impact that information providers participating in a persistent query system will incur.

From a systems perspective, PIE will require significant scalability to operate on a large scale. While we believe that PIE will scale well, this relies to some degree on the scalability of the middleware on which PIE is implemented. This implies that work on the CORBA ORB used to support the implementation should be considered especially in terms of the scalability, real-time, and fault-tolerance characteristics of such ORBs. While the implementation to date is pure Java, future implementations of the persistent query protocol even within PIE could be implemented in C or C++. Quantitative studies of the requirements placed on servers and on networks should be made to determine the computational

requirements for the system. Optimizations such as clustering queries, clustering user profiles, and a single VIRP subscriptions on behalf of all queries from a given PIE server might also be investigated. In addition, persistent querying is not required to run over CORBA, and advances in other middleware technologies should be noted as possible platforms for implementing additional PIE infrastructure. Additional work on the user interface might result in an HTML or XML web browser-based client.

In all, there is significant research to be done that ranges from systems work in scalability to fundamental questions about information retrieval techniques. All will benefit persistent querying, PIE, and SDI by creating systems that are more effective for users and usable by large numbers of participants, both users and information providers.

Bibliography

[AAB+98] Demet Aksoy, Mehmet Altinel, Rahul Bose, Ugur Cetintemel, Michael Franklin, Jane Wang, and Stan Zdonik. Research in Data Broadcast and Dissemination. In *Proceedings of the First International Conference on Advanced Multimedia Content Processing*, Osaka University, Osaka, Japan, November 1998.

[AAB+99] Mehmet Altinel, Demet Aksoy, Thomas Baby, Michael Franklin, William Shapiro, and Stan Zdonik. DBIS-Toolkit: Adaptable Middleware for Large Scale Data Delivery. In *Proceedings of the 1999 ACM SIGMOD International Conference on Management of Data*, pages 544–546, Philadelphia, PA, June 1999.

[AF00] Mehmet Altmel and Michael J. Franklin. Efficient Filtering of XML Documents for Selective Dissemination of Information. In *Proceedings of International Conference on Very Large Data Bases*, Cairo, Egypt, September 2000.

[Bal98] Marko Balabanovic. Learning to Surf: Multiagent Systems for Adaptive Web Page Recommendation. PhD Thesis SIDL-TR-98-1605, Computer Science Department, Stanford University, 1998.

[BC92] Nicholas J. Belkin and W. Bruce Croft. Information Filtering and Information Retrieval: Two Sides of the Same Coin? *Communications of the ACM*, 35(12):29–38, December 1992.

[BS95] Marko Balabanovic and Yoav Shoam. Learning Information Retrieval Agents: Experiements with Automated Web Browsing. In *Working Notes of the AAAI Spring Symposium Series on Information Gathering from Distributed, Heterogeneous Environments.*, 1995.

[BSY96] Marko Balabanovic, Yoav Shoham, and Yeogirl Yun. An Adaptive Agent for Automated Web Browsing. Technical Report SIDL-WP-1995-0023, Stanford Digital Library Project, Stanford University, February 1996.

[CDTW00] Jianjun Chen, David J. DeWitt, Feng Tian, and Yuan Wang. NiagaraCQ: A Scalable Continuous Query System for Internet Databases. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data*, Dallas, TX, May 2000.

[CGM97] Arturo Crespo and Hector Garcia-Molina. Awareness Services for Digital Libraries. In Carol Peters and Constantion Thanos, editors, *Research and advanced technoloty for digital libraries: First European converence; proceedings/ECDL 1997; In Lecture Notes in Computer Science*, volume 1324, pages 147–171, Pisa, Italy, September 1997. Springer Verlag.

[CRW98] Antonio Carzaniga, David S. Rosenblum, and Alexander L. Wolf. Design of a Scalable Event Notification Service: Interface and Architecture. Technical Report CU-CS-863-98, Department of Computer Science, University of Colorado, August 1998.

[DL00] J. R. Davis and C. Lagoze. NCSTRL: Design and Deployment of a Globally Distributed Digital Library. *Journal of the American Society for Information Science*, 51(3):273–280, March 2000.

[FV99] James C. French and Charles L. Viles. Personalized Information Environments: An Architecture for Customizable Access to Distributed Digital Libraries. *D-Lib Magazine*, 5(6), June 1999.

[FW91] H. P. Frei and M. F. Wyle. Retrieval Algorithm Effectiveness in a Wide Area Information Filter. In *Proceedings of the 14th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 114–122, Chicago, IL, October 1991.

[FZ97] Michael Franklin and Stanley Zdonik. A Framework for Scalable Dissemination-Based Systems. In *Proceedings of OOPSLA*, pages 94–105, Atlanta, GA, October 1997.

[GNOT92] David Goldberg, David Nichols, Brian M. Oki, and Douglas Terry. Using Collaborative Filtering to Weave an Information Tapestry. *Communications of the ACM*, 35(12):61–70, December 1992.

[Gro00] Object Management Group. *The Common Object Request Broker: Architecture and Specification.* Object Management Group, 2.4.1 edition, November 2000.

[GS97] Aniruddha S. Gokhale and Douglas C. Schmidt. Evaluating CORBA Latency and Scalability Over High-Speed ATM Networks. In *Proceedings of the ICDS*, Baltimore, MD, May 1997.

[GS98] Aniruddha S. Gokhale and Douglas C. Schmidt. Measuring and Optimizing CORBA Latency and Scalability Over High-speed Networks. *IEEE Transactions on Computers*, 47(4), April 1998.

[GW96] Andrew S. Grimshaw and Wm. A. Wulf. Legion – A View from 50,000 Feet. In *Proceedings of the Fifth IEEE International Symposium on High Performance Distributed Computing*, Los Alamitos, CA, August 1996.

[HF99] Annnika Hinze and Daniel Faensen. A Unified Model of Internet Scale Alerting Services. Technical Report TR-B-99-15, Freie Universitat, Berlin, Germany, 1999.

[Hou73] Edward M. Housman. Selective Dissemination of Information. In Carlos A. Cuadra and Ann W. Luke, editors, *Annual Review of Information Science and Technology*, volume 8 of *Annual Review of Information Science and Technology*, chapter 7, pages 221–241. American Society for Information Science, Washington, DC, 1973.

[HV99] Michi Henning and Steve Vinoski. *Advanced CORBA Programming with C++*. Addison-Weseley, 1999.

[KM91] B. Kahle and A. Medlar. An Information System for Corporate Users: Wide Area Information Servers. *Connexions - The Interoperability Report*, 5(11):2–9, 1991.

[LC98] R. Lasher and D. Cohen. RFC 1807: A Format for Bibliographic Records, September 1998.

[LG96] Michael J. Lewis and Andrew Grimshaw. The Core Legion Object Model. In *Proceedings of the Fifth IEEE International Symposium on High Performance Distributed Computing*, Los Alamitos, CA, August 1996.

[Lie95] Henry Lieberman. Letizia: An Agent That Assists Web Browsing. In *Proceedings of the International Joint Conference on Artificial Intelligence*, pages 924–929, Montreal, Canada, August 1995.

[LP97] Ling Liu and Carlton Pu. Dynamic Query Processing in DIOM. *IEEE Bulletin on Data Engineering*, 20(3):30–37, September 1997.

[LPBZ96] Ling Liu, Carlton Pu, Roger Barga, and Tong Zhou. Differential Evaluation of Continual Queries. In *IEEE Proceedings of the 16th International Conference on Distributed Computing Systems*, pages 458–465, Hong Kong, May 1996.

[LPT99a] Ling Liu, Carlton Pu, and Wei Tang. Continual Queries for Internet Scale

Event-Driven Information Delivery. *IEEE Transactions on Knowledge and Data Engineering*, 11(4):610–628, July/August 1999.

[LPT99b] Ling Liu, Carlton Pu, and Wei Tang. Supporting Internet Applications Beyond Browsing: Trigger Processing and Change Notification. In *Proceedings of the 5th International Computer Science Conference*, Hong Kong, December 1999. Springer Verlag.

[LPT00] Ling Liu, Carlton Pu, and Wei Tang. *WebCQ* - Detecting and Delivering Information Changes on the Web. In *Proceedings of the IEEE Conference on Information and Knowledge Management*, Washington, DC, November 2000.

[Luh58] H. P. Luhn. A Business Intelligence System. *IBM Journal of Research and Development*, 2(4):314–319, October 1958.

[Mae94] Pattie Maes. Agents that Reduce Work and Information Overload. *Communications of the ACM*, 37(7):30–40, July 1994.

[MD89] Dennis R. McCarthy and Umeshwar Daya. The Architecture of An Active Data Base Management System. In *Proceedings of the 1989 ACM SIGMOD International Conference on Management of Data*, Portland, Oregon, June 1989.

[PD99] Norman W. Paton and Oscar Díaz. Active Database Systems. *ACM Computing Surveys*, 31(1):63–103, March 1999.

[PL98] C. Pu and L. Liu. Update Monitoring: The CQ project. In *"Proceedings of the 2nd International Conference on Worldwide Computing and Its Applications"*, pages 396–411, 1998.

[POS+99] Irfan Pyarali, Carlos O'Ryan, Douglas Schmidt, Nanbor Wang, Vishal Kachroo, and Aniruddha Gokhale. Applying Optimization Principle Patterns

to Real-time ORBs. In *Proceedings of the Fifth USENIX Conference on Object-oriented Technologies and Systems (COOTS '99)*, San Diego, CA, May 1999.

[Pow00] Allison L. Powell. *Database Selection in Distributed Information Retrieval: A Study of Multi-Collection Information Retrieval.* PhD thesis, University of Virginia, Department of Computer Science, University of Virginia, Charlottesville, VA 22903, USA, December 2000.

[RD98] Satish Ramakrishnan and Vibha Dayal. The PointCast Network. In *Proceedings of the 1998 ACM SIGMOD International Conference on Management of Data*, page 520, Seattle, WA, June 1998.

[RW97] David S. Rosenblum and Alexander L. Wolf. A Design Framework for Internet-Scale Event Observation and Notification. In Mhedi Jazayeri and Helmut Schauer, editors, *Proceedings of the 6th European Software Egnineering Conference*, volume Software Engineering Notes of *Lecture Notes in Computer Science*, pages 344–360, Zurich, Switzerland, September 1997. ESCE / FSE, Springer-Verlag.

[Sch01] Douglas C. Schmidt. Evaluating Architectures for Multi-threaded CORBA Object Request Brokers. *Communications of the ACM, To Appear.*, 2001.

[SM93] Beerud Sheta and Pattie Maes. Evolving Agents for Personalized Information Filtering. In *Proceedings of the Ninth IEEE Conference on Artifical Intelligence for Applications*, pages 345–352, Orlando, FL, March 1993.

[TGNO92] Douglas Terry, David Goldberg, David Nichols, and Brian Oki. Continuous Queries over Append-Only Databases. In *Proceedings of the 1992 ACM SIGMOD International Conference on Management of Data*, pages 321–330, San Diego, CA, January 1992.

[WF89] M. F. Wyle and H. P. Frei. Retrieving Highly Dynamic, Widely Distributed Information. In N. J. Belkin and C.J. van Rijsbergen, editors, *Proceedings*

of the 12th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval, pages 108–115, Boston, MA, June 1989.

[WKLW98] S. Weibel, J. Kunze, C. Lagoze, and M. Wolf. RFC 2413: Dublin Core Metadata for Resource Discovery, September 1998.

[YGM94] Tak W. Yan and Hector Garcia-Molina. Distributed Selective Dissemination of Information. In *Proceedings of the Third International Conference on Parallel and Distributed Information Systems (PDIS 94), Austin, Texas, September 28-30, 1994*, pages 89–98, 1994.

[YGM95] Tak W. Yan and Hector Garcia-Molina. SIFT - A Tool for Wide-Area Information Dissemination. In *Proceedings of the 1995 USENIX Technical Conference*, pages 177–186, New Orleans, LA, January 1995.

[YGM99] Tak W. Yan and Hector Garcia-Molina. The SIFT Information Dissemination System. *ACM Transactions on Database Systems*, 24(4):529–565, 1999.