

# The Case for Proactive Directory Services

Fabián E. Bustamante      Patrick Widener      Karsten Schwan  
College of Computing  
Georgia Institute of Technology  
Atlanta, Georgia 30332, USA  
{fabianb, pmw, schwan}@cc.gatech.edu

## Abstract

Technology advances and increasing end-user expectations of distributed applications pose scalability challenges for directory services and consistency challenges for their clients. In particular, the number of objects, the number of attributes per object, and the rate of change of both objects and attributes are increasing. At the same time, context-aware applications such as active portals require that the data they obtain from directories be highly consistent. We argue that traditional pull-based interfaces to directory services are insufficiently scalable for both clients and servers, and propose a complementary *proactive* (push-based) interface. We describe the design and implementation of a directory service with such an interface, the Proactive Directory Service (PDS), and compare its performance against off-the-shelf implementations of DNS and the Lightweight Directory Access Protocol. Our experiments show that maintaining a high degree of consistency through a strictly pull-based interface may impose intolerable high loads on clients, servers, and the network, depending on the rate of change of the object(s) involved. By making the degree of consistency of the client data independent of the frequency with which it is updated, a proactive interface allows applications to obtain a perfect degree of consistency with a reasonable load on resources.

## 1 Introduction

The growth of the Internet, technology advances in wireless and wide-area communications, and increases in the capabilities of mobile computing devices have raised end user expectations concerning remote services. Namely, services should be available and accessible whenever needed and as appropriate for the contexts in which users currently operate. This has led to research on, amongst other topics, efficient service provision despite network dis- and re-connection [26], continuous reconciliation of partially consistent shared file state [33], and description-based discovery and location of services [1].

Our work concerns *directory services* used by distributed applications. Directories are employed to manage descriptive, fairly static, attribute-based information about a variety of objects, ranging from hostname/IP-address mappings to the office locations and telephone numbers of a company's employees. Directories typically manage data that is frequently read but rarely updated, and thus are optimized for reading, browsing and searching. Commonly used wide-area directory services include the Internet's Domain Name System (DNS) [24], DEC's Global Name Service (GNS) [22], and the X.500 Directory Service [29]; more recent proposals include the Intentional Naming System [1] and Active Names [34].

---

This work was supported in part by DARPA, by NSF grants C36-W63, ASC-9720167, CDA-9501637, CDA-9422033, and ECS-9411846, and by equipment donations from Sun Microsystems and Intel Corporation.

A key issue concerning directory and other network services is their scalability [30, 6]. Of particular concern to designers, implementors and users of directory services is the dramatic increase in the number of objects and object attributes referenced by distributed applications. For example, one major Internet Service Provider now manages customer and profile information for over 4 million subscribers [9]; even if a particular distributed application only “touches” 25% of them, a conservative estimate of 10 attributes per object implies the need to manage on the order of 10 million information items. The resulting scalability problems are exacerbated when directories are used in ubiquitous settings, where object attributes like end user location and operating context can change frequently, as experienced with mobile and wearable devices in automobiles or carried through buildings or homes [38, 8]. This is especially so with applications that must remain aware of certain changes to directory data but have little ‘intuition’ about the frequency or scope of those changes.

Our research contributes one key element of solutions to directory service scalability: *proactivity*. Current directory services typically provide an inactive client interface, where clients interested in the values of certain object attributes must explicitly request such information from the server. When object attributes are frequently updated, clients must query servers at a sufficiently high rate to maintain acceptable levels of consistency between their cached information and the actual values stored by servers. The basic idea behind of *proactivity* is to extend a directory service with dynamically customizable publish/subscribe interfaces. Such interfaces are used to notify clients about the changes in objects and object attributes currently of interest to them, with notification rates customized to individual clients’ needs. The result is that clients ‘know about’ the different attributes of interest to them at the levels of consistency that match their current needs.

Proactivity is not a replacement for the traditional request/reply or ‘pull-based’ model of client-server interaction, but an extension to it that is particularly beneficial for environments with frequently changing object attributes. Specific examples include those of location-aware services [20] and context-aware applications [8]. One such type of application being constructed by our group is *active portals*, these are portals whose entries and characteristics change according to current user location, contexts, and needs. Such portals are among the more demanding clients of future object and attribute directory services. We describe the construction of these types of applications, and how they make use of distributed directory services in Section 4.

In this paper we show analytically and demonstrate experimentally that (a) an inactive model of client-server interaction restricts the scalability of directory services, and (b) a proactive approach helps to circumvent this restriction. We do this by comparing the performance results attained with our *Proactive Directory Service*, a fully functional prototype implementing proactivity, with those attained with off-the-shelf implementations of the Domain Name System (DNS) and the Lightweight Directory Access Protocol (LDAP). By making the degree of consistency of the client data independent of the frequency with which it is updated, a proactive interface allows applications to obtain a perfect degree of consistency with reasonably low loads on resources. The utility of PDS and performance results attained with it both demonstrate the need for and benefits of proactivity.

The remainder of this paper is organized as follows. Section 2 describes our ideas in more detail and presents the design and architecture of the Proactive Directory Service. Section 3 discusses the performance and scalability benefits to be gained by proactivity and presents experimental results validating our hypothesis. Section 3.5 describes potential problems with proactivity and explains how we address them in PDS. A potential class of applications for proactivity is presented in Section 4. We discuss related work in Section 5, and conclude in Section 6.

## 2 A proactive approach

The Internet provides a distributed environment apt for many classes of applications including science, education, commerce and entertainment. As such applications extend from LANs to Wide-Area Networks, the number of objects they use increases dramatically [12, 17, 35]. Compounding this increase is growth in the number of object attributes of interest to applications. For instance, context-aware applications [31] exploit information about their execution environments in an attempt to improve performance and/or enhance usability. Here, context is defined as the collection of entities, and their attributes, considered relevant to the ongoing interaction between users and an application [8]. Relevant context information includes the location of use, the collection of nearby people, and the set of resources and their availability. Location information is not restricted only to spatial coordinates; it can also be environmental data such as lighting, noise level, and network connectivity, or even social information (whether you are talking to your advisor or another grad student).

The requirements of these types of applications not only translate into a substantial increase in the number of objects and object attributes managed by directory services, but they also result in a significant increase in the frequency with which the values of those attributes change. All directory services known to us provide an exclusively pull-based client interface. With such an interface, clients have no option but to explicitly and repeatedly request information from the server. These requests must be made at a frequency sufficiently high to maintain an acceptable level of cache consistency. This model may suffice with a relatively low number of objects and attributes, and with attribute values that change infrequently, but it is not sufficient for handling the requirements of new applications.

A proactive approach to directory services can help address these issues. Specifically, we integrate proactivity into a directory service by associating channels for change notification with each object managed by the directory, and by allowing clients to subscribe to them as an alternative way of finding out about changes to their attributes. Changes are reported to interested parties over notification channels in the form of events. Sample events include:

- creation or removal of an entry in the directory service;
- binding an entry in the directory service to a name, or unbinding an entry from a name;
- changes to an existing entry in the directory; and
- changes to the attributes of an entry in the directory.

Different data is associated with each of these events. For example, it makes sense for a creation or change event to also carry the entry itself. However, this is not necessary for a removal event, as clients presumably have the last value of the entry and so re-distributing it would be redundant.

### 2.1 PDS design requirements

We identify the following requirements for a proactive directory service:

- **Performance.** It must provide acceptable performance. This does not mean that a proactive directory must necessarily outperform traditional directory services in areas such as search speed or transmission latency. Rather, its traditional (pull-based) interface should not exhibit unreasonable performance when compared to other traditional services, and its implementation of proactivity must not impose significant overhead.

- **Scalability.** It must be able to track an increasing number of objects, as well as an increasing number of attributes per-object, with performance degradation proportional to the increase(s).
- **Dynamic data.** Some of the data required by applications is highly dynamic: for example, network availability, host load, or physical location information. The service must be able to make this data available in a timely fashion.
- **Extensibility.** There is no easy way to define a single data model that will suit the needs of a wide range of applications. Hence, the service must be able to incorporate modifications to its underlying data model.
- **Customizable.** The ability to customize the service on a per-client basis becomes far more important in a proactive environment; clients that cannot predict when they will receive new information cannot prepare for its arrival in the same way pull-based clients can.

Details of the design and implementation of the Proactive Directory Service (PDS) are described next.

## 2.2 Basic PDS architecture

The directory service provided by PDS is implemented as a collection of cooperating objects. *Domains* implement namespaces using *Contexts*, which are used to bind user-meaningful names to *Entities*.

**Domains.** We use the term *Domain* differently than does DNS, although our reasons for using it are similar to those of DNS. A single secular namespace does not scale well, and presents frustrating challenges (versioning and garbage collection, for example) to designers of different applications. While DNS domains are collections of hosts under common administration, a PDS domain implements an individual namespace. Domains themselves are distinguishable in a fairly flexible manner; they are identified by name (a string), type (string), version number (integer), and owning application (string). Any combination of these values can be used to retrieve a single domain or a collection of domains from the server.

**Contexts and namespaces.** Each domain implements its namespace using *contexts*. The namespace is implemented roughly as a tree, with a context at each vertex; the domain managing a particular namespace owns a single “root” context for it. Each context contains a list of name-to-entity bindings. Contexts themselves may be bound to names in other contexts. In this manner, a general directed naming graph is maintained, with arcs from contexts to contexts or entities labeled with names<sup>1</sup>.

Internally, name resolution is accomplished by traversing the tree of contexts. PDS provides an obvious and intuitive hierarchical naming interface to the client. Similar to that provided by common filesystems, names are strings in which name components are separated by distinguished separator characters. Beginning at the root context, each name component represents the path to another context.

Entities and contexts may be bound to names in more than one context and to more than one (different) name in the same context. From a client perspective, entities are unaware of the names to which they are bound.

**Entities.** The basic element in PDS is an *entity*, which is an identifiable “thing” in the real world. Notice that an entity is not the object itself but a representative of it – an avatar. Each entity has a particular set of properties, or *attributes*, and each attribute has a value. Specific entities can be located in the directory by providing a name which is resolved by the owning Domain. Entities or collections of entities can be retrieved

---

<sup>1</sup>We currently ignore the issue of cycles in the naming graph.

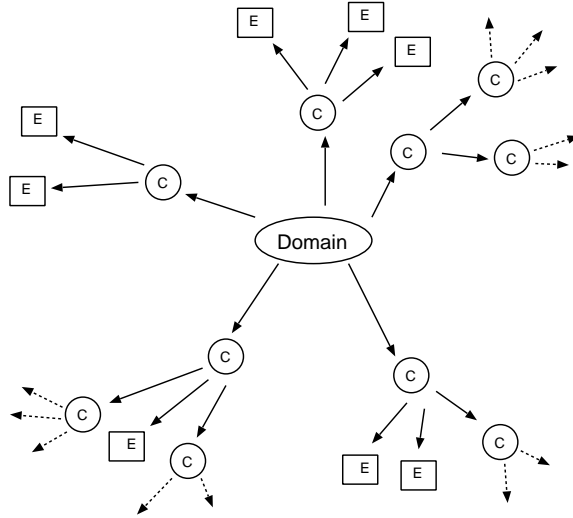


Figure 1: **Domains, contexts (C) and entities (E).** Each Domain owns a root Context in which all descendants are named.

by providing a set of attributes for use as a search template; all entities matching the set of attributes are returned.

### 2.3 Achieving proactivity

Each of the objects (domains, contexts, and entities) managed by PDS has associated with it an event channel. PDS interface allows clients to subscribe to the event channel for a particular object. Clients provide an event handler which is called by the communications system when new events arrive. Each of the state-changing operations implemented by PDS submits an event to the channel of the appropriate object. When the owner of an object changes the value of some of the object's attributes (the network address of a machine providing a particular service, for example), a change event is submitted to the event channel belonging to the entity representing the object.

### 2.4 Other architectural features

This section describes other PDS architectural features of interest. A detailed description of PDS can be found in [39].

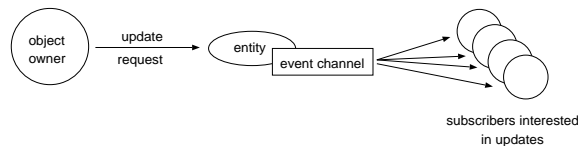


Figure 2: **Achieving proactivity: when the owner of an object updates any of the object's attributes, notifications are sent to all interested clients.**

**Identifying objects.** Entities are named with respect to a particular domain; recall that each domain is a separate namespace. Operations on entities, therefore, require both an identifying name and a specification of the domain in which the name should be resolved. Domains are uniquely identified by a set of primitive-type items. Specifying this set of items for each entity operation is cumbersome and requires a lookup at the server to identify the domain object in question. To avoid this, PDS provides *objectIDs*, which are opaque-type identifiers that uniquely and globally (across PDS instances) identify a PDS object. ObjectIDs can be transmitted “out-of-band” between application programs to identify PDS objects, or even stored in PDS for retrieval by other clients. Any PDS domain, context, or entity object is identifiable by its objectID.

**Heterogeneous architecture support.** Directory services like DNS and X.500 provide architecture-independent interoperability by a lowest-common-denominator approach. All data is represented and transmitted in ASCII. The simple goals of DNS and the comparative simplicity of its data model make this an appropriate choice. Services whose data model is extensible in arbitrary fashion (like X.500, or XML-based services) suffer in performance, however, due to ASCII marshaling and unmarshaling. PDS avoids this performance penalty by relying on a fast binary encoding scheme [10] (see Figure 8). This scheme outperforms other comparable binary data-transport methods, and allows PDS to provide an extensible data model while still using binary-format data transmission [3].

## 2.5 Discussion and comparison

Proactivity is a well-established system design technique. Physical devices such as buses and disks use interrupts as a proactive means of informing operating system kernels that a state change has occurred. This allows kernels to avoid repeated polling for state changes. Write-through caches are ‘proactive’, in that they ensure cache consistency without resulting in cache faults for applications that use such data. The publish/subscribe paradigm of distributed system design, with its roots in non-distributed reactive programming models, is another instance of proactivity, because peers in a publish/subscribe or event-passing system proactively notify interested parties of state changes.

Proactivity is also commonly used to integrate component-based software systems [28, 32], as those constructed with Enterprise Java Beans, Microsoft’s Component Object Model and its distributed extension (DCOM), and the developing specification of the CORBA Component Model (CCM) in OMG’s CORBA version 3. Specifically, a common technique for integrating the different components of a system is *event-based* invocation, also known as implicit or reactive invocation, which has historical roots in systems based on actors [21], daemons, and packet-switched networks. Event-based integration is attractive as it strongly supports software reuse and facilitates system evolution [15, 14, 11].

The use of proactivity in directory services has some precedents. Until recently, changes to a DNS zone (DNS zones are contiguous subdivisions of a namespace) were propagated among the interested (replicated) name servers through a pulling mechanism initiated by the replications. In order to reduce the load imposed on the master name servers, longer refresh times of the zone’s data were normally adopted, but that benefit would come at the cost of long intervals of incoherence among authority servers whenever the zone is updated. In response, RFC 1996 [37] proposed a mechanism for prompt notification of zone changes (DNS NOTIFY) through a proactive approach. Master servers can now inform slave servers when the zone has changed through an interrupt “which it is hoped will reduce propagation delay while not unduly increasing the masters’ load”. PDS as described in the previous section takes steps toward proactivity beyond this by communicating the updates themselves to its interested clients where the data is actually used.

In the following section we describe and quantify the benefits gained from using proactivity in directory services.

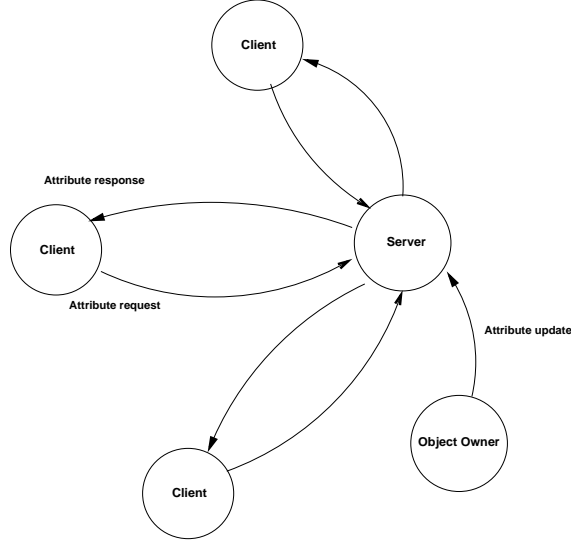


Figure 3: Modelling clients, servers and entities' owners.

### 3 Benefits of proactivity

In this section we validate our proactive approach by contrasting the scalability of PDS with that of off-the-shelf implementations of DNS and LDAP. While PDS provides pull- and push-based interfaces, both of these systems follow a strictly pull-based approach. BIND DNS is used because it is a highly optimized (primarily for reading) system; OpenLDAP provides functionality and extensibility similar to PDS. Before presenting the results of our experimental comparison, we evaluate both approaches analytically.

#### 3.1 Analytical comparison

We can analyze the load imposed by pull-based and push-based model directory interfaces through a simple model. A directory server manages a number of entities on behalf of a their owners. Many clients want to keep track of different subsets of those entities (Figure 3).

Assume a fixed period of time over which the owner of an entity in the directory makes a series of updates, we wish to calculate the number of client requests that would be necessary to maintain different degrees of consistency over that period.

Let  $U$  be a set of updates,  $u_i$ , occurring over a period  $T$  at some random instants of time, and let  $p$  be the frequency at which a client pulls a non-proactive server. The total time over which the client will have an inconsistent copy can be computed as:

$$I = \sum_{i=1}^U d_i = \sum_{i=1}^U (p - u_i \bmod p)$$

i.e. the client's degree of consistency, or the percentage of time at which the client will have a consistent copy, is:

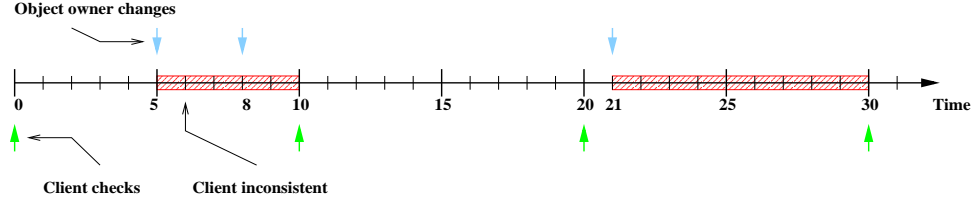


Figure 4: **Timeline of client's periods of inconsistency.**

$$(T - I)/T * 100\%$$

Notice that the degree of consistency depends on the frequency at which the client requests an update from the server and the rate at which the owner of the entity updates its attributes.

For example, if over a 30 hour time period the entity owner changes the object value at hours 5, 8, and 21 while the client polls the server for this object value every 10 hours, the client will have the wrong value (5 + 9) hours over those 30, what corresponds to an accuracy of  $(30 - 14) / 30 = 0.53$ , or 53%. This can be graphically represented as in Figure 4.

As the degree of consistency is determined by the client frequency of pulls and the entity owner rate of updates so is the number of messages exchanged between the client and the server (and so the load on resources) required to keep a given degree of consistency. Figure 5 shows the number of messages between the resolver and the directory server, required to maintain a given level of consistency for the updates and pull frequency quoted in Figure 6.

In contrast, a proactive interface allows applications to obtain a perfect degree of consistency at a reasonable load on resources by making the degree of consistency of the client data independent of the update frequency of the owner of it. While, for this example, over 800 messages are required to obtain a perfect degree of consistency without proactivity, with a proactive interface only 15 are needed!

### 3.2 Performance metrics

In the next three sections we detail our experimental comparison of the two approaches. We first describe the metrics used to quantify the benefits of our proactive approach. After that we describe our experimental setup and present the results of the actual comparison.

Intuitively, proactivity provides scalability and high performance by reducing the amount of work done by clients (and correspondingly by servers) in order to become aware of directory updates. Stated more precisely, the objective of a directory service is to attain high degrees of consistency across the information stored in the directory and in clients. The attainment of consistency requires message exchanges between server and clients. The following metrics capture these facts:

- **Degree of consistency:** the percentage of time that a client has the correct value of a directory entry. While pull-based clients must poll the server with increasing frequency to increase their degree of accuracy, clients of proactive servers are always 100% accurate (because they automatically receive updates when changes are made at the server).
- **Number of client requests required:** the number of client requests that is required for a client to maintain a particular degree of consistency.

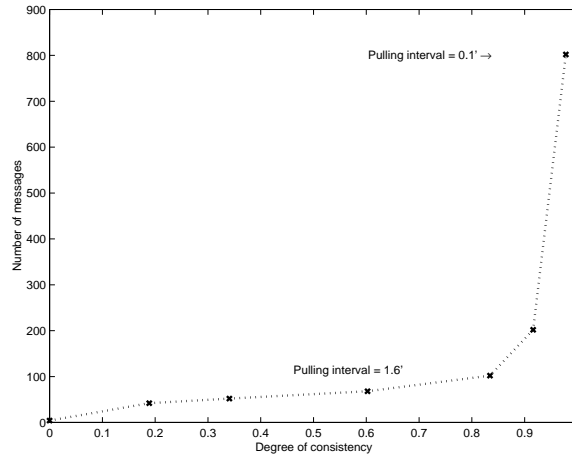


Figure 5: Number of messages required to achieve a given degree of consistency.

```

Length of period: 40 minutes
Number of updates by owner: 15
Times of updates:
Update 1      2      3      4      5
Time  2.673  3.678  15.336  15.340  16.64

Update 6      7      8      9      10
Time  20.777  21.188  25.265  26.061  26.1568

Update 11     12     13     14     15
Time  26.846  28.906  30.257  35.388  37.218

Number of resolver's pull intervals: 7
Pull intervals:
Period      1  2  3  4  5  6  7
Pull interval 0.1 0.4 0.8 1.2 1.6 2.0 40.0

```

Figure 6: Owner's updates times and client's pull intervals.

- **CPU load on client:** the processing required by the client to send a certain number of requests.
- **CPU load on server:** the processing required at the server to respond to requests by the client.

Proactivity reduces the load on the server by significantly reducing the number of client requests for updates. Client load is reduced because the server (or the object's owner) is responsible for notifying the client when changes occur to the object. The number of messages in the system is reduced by eliminating client polling for updates, resulting in an optimal message-per-update. These intuitive statements are validated by experimentation described next.

### 3.3 Experimental environment

The experiments presented in this section compare the performance of BIND DNS 8.2.2-7 [5], OpenLDAP 1.2.11 [13], and PDS. We configure one host each as client, server, and object's owner. Each host used in our test has 4 Intel Pentium Pro processors with 512MB of RAM, runs RedHat Linux 6.2, and is connected with the other hosts by a 100Mbps Fast Ethernet.

### 3.4 Performance comparisons

We generated a number of updates occurring at randomly generated intervals over a fixed period of time (see Figure 6). Since the actual loads on resources incurred on a particular situation depend on such a wide variety of variables that we cannot characterize the entire performance space, we decided to take an illustrative example and examine it.

Given the sequence of owner's updates and the set pull frequencies quoted in Figure 6, we measured the load imposed on client and server for the DNS, LDAP, and PDS implementations using their pull-based interface, and by PDS using its push-based interface. All client-side caches were disabled, and we ran DNS over TCP (by setting its 'vc' option) for purposes of comparison. Figure 7 and Figure 8 show the client and server loads for all four cases we studied.

Figure 7 shows that with the PDS proactive interface, a perfect (1) degree of consistency can be obtained, at a load on the client that is one-fourth that of DNS and half that of LDAP.

Figure 8 compares the load imposed on each server by our experiment. It shows that a perfect degree of consistency can be obtained at a reasonably low server load with a proactive interface. The scalability problem of a pull-based interface, as faced by the DNS and LDAP implementations, are clear (our implementation, although better than those, shows a similar trend). Notice, however, that the load imposed on these servers is not significantly serious. Even at the point of perfect consistency (highest-rate of pull for pull-based clients), our experiment imposes a load of 400 requests/replies over 6 minutes, i.e. only 1.11 messages per second.

The experiments demonstrate that an inactive model of client-server interaction restricts the scalability of directory services, and state the clear benefits of a proactive approach.

### 3.5 The need for customization

The primary advantage to clients of pull-based interfaces is control. Pull-based interfaces allow clients to control when/if messages are sent and to anticipate replies (since the fact that a reply is impending and the type of information the reply carries are both known). Proactivity allows clients to trade control for performance, as message traffic is only generated when updates occur. As long as updates occur infrequently, this lack of control is not significant. However, a client that registers interest in an object that begins changing

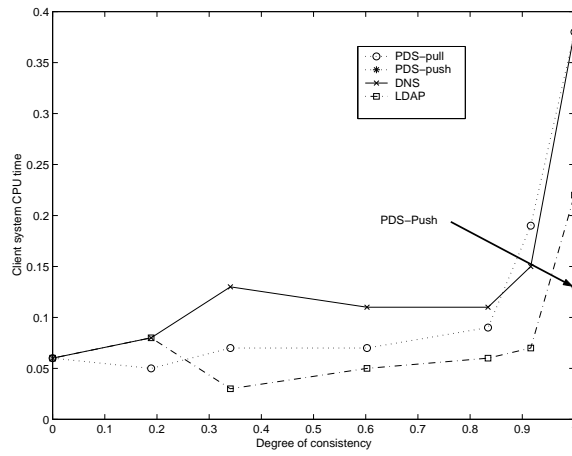


Figure 7: Client load (as measured by system CPU time) needed to reach a given degree of consistency.

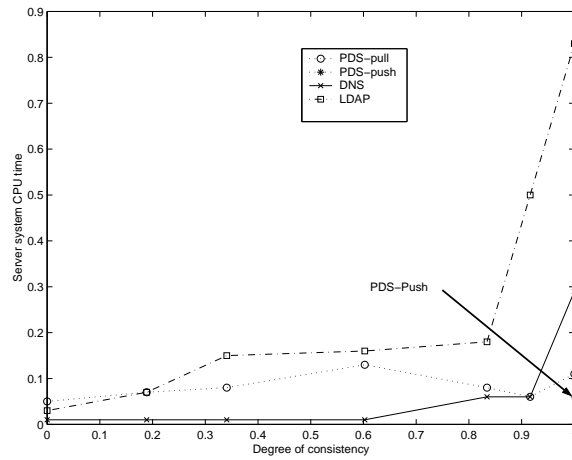


Figure 8: Server load (as measured by system CPU time) needed to reach a given degree of consistency.

```

{
    if ((input.balance < 120) || (input.trade_price > 5000)) {
        return 1; /* submit event into notification channel */
    }
    return 0; /* do not submit event into notification channel */
}

```

Figure 9: **A specialization filter that passes only balance reports outside a pre-defined range.**

with unanticipated frequency soon finds itself swamped with update message. These update messages may not even be needed when they arrive, or may only be needed depending on other application-specific factors; proactivity in this case does more harm than good.

At first glance, providing a filter at the client to discard unwanted or unneeded updates might seem beneficial. Although this does allow the application to ignore updates, the update messages are still sent across the network, increasing the load on the server, the network, and the client. Providing a single interface at the server to control proactive traffic is insufficient, as different clients interested in changes may have different criteria for discarding update messages.

A better approach allows client-specific *customization* of the update channel. To customize a channel, a client provides a specification of what events it will be interested in. The server then uses these specifications, on a per-client basis, to determine whether or not to send the update event.

PDS uses an event infrastructure that directly supports this type of behavior. The ECho event transmission system allows clients to *derive* customized channels from existing ones by supplying a filter function [11]. This filter function is written in a portable subset of C and is dynamically compiled and installed at the server. Filter functions have rich degrees of expressiveness; it is possible to examine the event data in the filter function to determine whether or not the event should be sent. For example, an event corresponding to an update in the value of a room sensor such as temperature may only be useful to an application if the value of the temperature is above a certain number. An interested client could then customize the update channel so that updates would only be transmitted from the server if that condition is satisfied. Note that this customization is per-client; the actions and interests of one client do not affect updates received by another.

The cost/benefit analysis of customization is not entirely straightforward, however. Previous research [11] has indicated that in some cases the cost of implementing the customization outweighs the benefits (in terms of network traffic and CPU load).

## 4 Active Portals: An application

In this section, we describe active portals and their use of distributed directory services.

Internet application providers increasingly seek to establish themselves as *portal services*. Portals are dedicated World Wide Web locations provided to customers through which any information on the Internet is available, and to which targeted information (based on demographic or user-selected criteria) is delivered. The ability to customize or personalize portals based on properties of each user is increasingly a means for businesses to distinguish their offerings from those of their competitors. The domain of properties useful for customization is large and diverse. Physical location of users provides the basis for determining which weather forecast and which traffic congestion reports to display. Personal preferences determine which sections of the online version of the user's hometown newspaper to retrieve. Characteristics of the network connection like observed latency and bandwidth determine what quality of streaming audio and video can be supported. There are many other properties that can be used for similar customization purposes.

Such context-aware web applications can keep track of their contexts' properties through a proactive directory service. Introducing this level of indirection into the customization process relieves the user of having to provide all desired properties each time she visits the portal.

Problems arising for portals and the directory services on which they rely include:

1. *stale directory information*, which means that the information in the directory does not reflect the actual state of the user; and
2. *inconsistent information* across directory and portal services, which means that updates made to the directory have not yet been reflected to the portal.

Information staleness and inconsistency are not problematic when user profiles are relatively static, as one might argue for current usage of Internet portals. In addition, unless users depend on information correctness in portals, the effects of inaccurate information may merely be annoying, not disastrous. In contrast, our efforts are to construct *active portals*, which are portals that reflect current user state and desires even when both change frequently. Active portals are useful in many domains, the one targeted by our efforts being mobile users who need to access information not only based on current location but also on the current context in which they are operating.

The environments in which we are deploying active portals are (1) the Georgia Tech Aware Home and (2) ubiquitous, "in field" systems. In the case of the home, context may be established by the room in which the user is currently located, or by actions taken by the user, such as sitting down at a conference table with other home occupants. In ubiquitous systems, context includes location, but equally relevant are factors like the current wireless bandwidth available to certain end user. In both cases, unless the active portal has up-to-date knowledge of user context, it will fail to provide appropriate services to the user, sometimes with disastrous effects. For example, in a ubiquitous system setting, a portal may misguide a user driving a car by failing to inform him about an upcoming traffic accident, or it may fail to direct the user of a PDA to the proper soccer field for the upcoming youth soccer match (e.g., the portal may not know about the fact that the match's field was recently changed). In the aware home environment, the portal cannot provide data about other conference participants (and their current actions) when the portal user sits down at the table, thereby making it difficult to strike up a conversation like: "did you figure out what was wrong with the camera in the kitchen?". In all such cases, portals must actively seek to renew their information content, as per the accuracy and timeliness required by their end users.

## 5 Related work

There are many variants on the common theme of directory services. Classical directory services tend to contain descriptive, attribute-based information on a variety of objects but, generally, without supporting complicated transactions or roll-back schemes as those found in traditional database management systems. However, none support the ability to proactively notify clients of updates as in PDS.

Grapevine [2] and Global Name Service [22] were two pioneering distributed directory services. Other more recent services such as DNS [25] and the X.500 Directory Service [29], provide such services under the assumption of fairly stable mappings between objects' attributes and their values. Such an assumption allows them to make heavy use of caching to improve look-up performance and service scalability. However, they are unsuited for applications in which updates occur with even moderate frequency. DNS, however, is not extensible in a general manner. X.500 is highly complex, requiring an ISO protocol stack and ASN.1 data encoding. LDAP (Lightweight Directory Access Protocol) is a streamlined version of the X.500 directory service. It removes the requirement for an ISO protocol stack, defining a standard wire protocol based on

the IP protocol suite. It also simplifies the data encoding and command set of X.500 and defines a standard API for directory access.

Other more recent research includes the Intentional Naming System (INS) [1] which integrates name resolution and routing. INS allows clients to send messages by describing the attributes of the destination rather than its location on the network. Active Names [34] maps remote service names to chains of mobile programs responsible for locating the remote objects and transporting back the response. Both Active Names and INS concentrate on the problem of efficient and flexible name-to-object resolution, as opposed to PDS' emphasis on providing low-impact client consistency. Each of these objectives is desirable in a wide-area environment, and the concept of proactivity is certainly compatible with either Active Names or INS. The approach to object naming employed in Semantic File Systems [16] is similar to that taken by INS.

Directory services supporting entries with attributes have made feasible service and resource discovery by processing queries containing a set of desired attributes. Jini [23], the Service Location Protocol (SLP) [27, 36], and the Service Discovery Service [7] all provide attribute-based service discovery for heterogeneous devices. PDS also supports retrieval of entities based on attributes. Jini provides services using Java RMI as a transport mechanism; PDS uses a fast binary transport encoding mechanism that provides superior performance. PDS does not address issues of authentication and secure communication as does SDS. The proactive approach of PDS would be complementary to any of these services.

Our work on PDS has some similarities with research on maintaining cache consistency. Gwertzman and Seltzer [19] provide a detailed analysis of World Wide Web cache consistency approaches using trace-driven simulation. Cao [4] reports the advantages of proactivity in maintaining cache consistency between Web browsers and servers; consistency is maintained by servers proactively notifying clients (browsers) of changes through invalidation messages. The clients are still responsible for retrieving the updated web page from the server. In contrast, PDS has the ability to supply the updated object/attribute data in the update message, thereby saving a message exchange. We affirm the conclusions presented in both papers about the impact of client polling on server load and network usage. It is worth noting, however, that the consistency requirements of distributed applications using a directory service such as PDS may be more stringent than those of the typical Web user.

Other unique approaches utilizing proactivity exist. Gwertzman and Seltzer introduces HTML page replication between servers as a method of exploiting page access history for increased cache effectiveness [18]. As stated above, PDS is designed for application usage where consistency requirements may be more stringent, and we have not yet explored replication issues. Heidemann and Shah proposes an extension to the *cron* utility (called *lcron*) which uses proactive notification of changes in geographic location to trigger user-defined system changes [20]. This usage of proactivity is more in the spirit of the notifications required by active portals, although *lcron* does not function in a distributed manner.

## 6 Conclusion and future work

In this paper, we have described how a proactive directory service can facilitate the construction of context-aware scalable distributed applications and services such as Active Portals and others. These systems require clients to maintain high degrees of consistency with their directory servers. We have constructed a proactive directory service, PDS, and compared both its traditional, pull-based interface and its proactive interface with those of DNS and LDAP. Our results indicate that proactive directory services make complete consistency possible while imposing tolerable loads on the client and server hosts. We also found that the use of pull-based interfaces results in comparably high amounts of server load, client load, and network messages, while still not providing complete consistency.

As it stands, PDS is nowhere near a complete directory service. Indeed, we did not set out to build a

complete service, as our focus has been on integrating proactivity into a directory framework. The issues involved in building a complete service (distributed name resolution such as that done by Active Names and INS, and replication and distribution of data, for example) have been well explored by other researchers, and we intend to incorporate some of the lessons they have learned. In particular, we intend to examine issues surrounding the use and definition of naming schemas in PDS, as well as methods by which our proactive architecture can make possible flexible replication strategies.

## Acknowledgments

Greg Eisenhauer participated in many discussions about the design both of PDS itself and of the experimental approach we have used. Mustaque Ahamad gave us valuable feedback on the similarities between our work and more general research on distributed cache consistency. Neil Bright assisted us with the configuration complexities of DNS.

## References

- [1] William Adjie-Winoto, Elliot Schwartz, Hari Balakrishnan, and Jeremy Lilley. The design and implementation of an intentional naming system. In *Proceedings of the 17th ACM Symposium on Operating System Principles*, pages 186–201, Kiawah Island, SC, December 1999. ACM.
- [2] Andrew Birrell, Roy Levin, Roger M. Needham, and Michael D. Schroeder. Grapevine: An exercise in distributed computing. *Communication of the ACM*, 25(4):260–274, April 1982.
- [3] Fabian Bustamante, Greg Eisenhauer, Karsten Schwan, and Patrick Widener. Efficient wire formats for high performance computing. In *Proc. of Supercomputing 2000 (SC 2000)*, Dallas, TX, November 2000.
- [4] Pei Cao and Chengjie Liu. Maintaining strong cache consistency in the world wide web. *IEEE Transactions on Computers*, 47(4):445–457, April 1998. Published in the 17th IEEE International Conference of Distributed Computing.
- [5] Internet Software Consortium. Bind domain name service. <http://www.isc.org/products/BIND/bind8.html>.
- [6] Inktomi Corporation. The Inktomi technology behind HotBot. <http://www.inktomi.com>, May 1996.
- [7] S. Czerwinski, B. Zhao, T. Hodes, A. Joseph, and R. Katz. An architecture for a secure service discovery service. In *Proceedings of ACM/IEEE MOBICOM*, pages 24–35, August 1999.
- [8] Anind K. Dey. Understanding and using context. *Personal and Ubiquitous Computing*, 5, 2001. Forthcoming Issue.
- [9] Earthlink, Inc. Earthlink homepage. <http://www.earthlink.com/>.
- [10] Greg Eisenhauer. Portable self-describing binary data streams. Technical Report GIT-CC-94-45, College of Computing, Georgia Institute of Technology, 1994. (*anon. ftp from ftp.cc.gatech.edu*).
- [11] Greg Eisenhauer, Fabian Bustamante, and Karsten Schwan. A middleware toolkit for client-initiated service specialization. In *Proceedings of the PODC Middleware Symposium*, July 2000.
- [12] Ian Foster and C. Kesselman. Globus: A metacomputing infrastructure toolkit. *International Journal of Supercomputer Applications*, 1997.
- [13] OpenLDAP Foundation. The OpenLDAP Project. <http://www.openldap.org/>.

- [14] David Garlan and David Notkin. Formalizing design spaces: Implicit invocation mechanisms. In *VDM'91: Formal Software Development Methods*, pages 31–44. Springer-Verlag, LNCS 551, October 1991.
- [15] David Garlan and Mary Shaw. An introduction to software architecture. In V. Ambriola and G. Tortora, editors, *Advances in Software Engineering and Knowledge Engineering, Volume I*. World Scientific Publishing Company, New Jersey, 1993.
- [16] D. Gifford, P. Jouvelot, M. Sheldon, and J. O'Toole. Semantic file systems. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles*, pages 16–25, October 1991.
- [17] Andrew Grimshaw, Adam Ferrari, Fritz Knabe, and Marty Humphrey. Legion: An operating system for wide-area computing. *IEEE Computer*, 32(5):29–37, May 1999.
- [18] James Gwertzman and Margo Seltzer. The case for geographical push-caching. In *Proceedings of the 1995 Workshop on Hot Operating Systems*, pages 51–55, May 1995.
- [19] James Gwertzman and Margo Seltzer. World-wide web cache consistency. In *Proceedings of the 1996 USENIX Technical Conference*, January 1996.
- [20] John Heidemann and Dhaval Shah. Location-aware scheduling with minimal infrastructure. In *Proceedings of the 2000 USENIX Technical Conference*, June 2000.
- [21] C. Hewitt. Planner: A language for proving theorems in robots. In *Proceedings of the First International Joint Conference in Artificial Intelligence*, 1969.
- [22] Butler W. Lampson. Designing a global name service. In *Proc. 4th ACM Symposium on Principles of Distributed Computing*, pages 1–10, Calgary, Alta Canada, August 1986. ACM.
- [23] Sun Microsystems. Jini. <http://java.sun.com/products/jini/>.
- [24] P. Mockapetris. Domain names - concepts and facilities. RFC 1034, Network Working Group, November 1987.
- [25] P. Mockapetris and K. J. Dunlap. Development of the domain name system. In *Symposium proceedings on Communications architectures and protocols (SIGCOM)*, pages 123–133, Stanford, CA, August 1988. ACM.
- [26] L.B. Mummert, M.R. Ebling, and M. Satyanarayanan. Exploiting weak connectivity for mobile file access. In *Proceedings of the 15th ACM Symposium on Operating System Principles*, pages 143–155, Copper Mountain, CO USA, December 1995.
- [27] C. Perkins. Service location protocol white paper. [http://playground.sun.com/srvloc/slp\\_white\\_paper.html](http://playground.sun.com/srvloc/slp_white_paper.html), May 1997.
- [28] Dewayne E. Perry and Alexander L. Wolf. Foundations for the study of software architecture. *ACM SIGSOFT Software Engineering Notes*, 17(4), October 1992.
- [29] S. Radicati. X.500 directory services: Technology and deployment. Technical report, International Thomson Computer Press, London, UK, 1994.
- [30] Yasushi Saito, Brian N. Berhad, and Henry M. Levy. Manageability, availability and performance in porcupine: a highly scalable, cluster-based mail service. In *Proceedings of the 17th ACM Symposium on Operating System Principles*, pages 1–15, Kiawah Island, SC, December 1999. ACM. Use of proactivity for soft state reconstruction.

- [31] Bill N. Schilit and Marvin M. Theimer. Disseminating active map information to mobile hosts. *IEEE Network*, 8(5):22–32, September 1994.
- [32] Mary Shaw and David Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice-Hall, 1996.
- [33] Douglas B. Terry, Marvin M. Theimer, Karin Petersen, Alan J. Demers, Mike J. Spreitzer, and Carl H. Hauser. Managing update conflicts in bayou, a weakly connected replicated storage system. In *Proceedings of the 15th ACM Symposium on Operating System Principles*, pages 172–183, Copper Mountain, CO, December 1995.
- [34] Amin Vahdat, Michael Dahlin, Thomas Anderson, and Amit Aggarwal. Active names: flexible location and transport of wide-area resources. In *Proceedings of USENIX Symp. on Internet Technology & Systems*, October 1999.
- [35] M. van Steen, P. Homburg, , and A.S. Tanenbaum. Globe: A wide-area distributed system. *IEEE Concurrency*, 7(1):70–78, January-March 1999.
- [36] J. Veizades, E. Guttman, C. Perkins, and S. Kaplan. Service location protocol. RFC 2165, Network Working Group, June 1997.
- [37] P. Vixie. A mechanism for prompt notification of zone changes (dns notify). RFC 1996, Network Working Group, August 1996.
- [38] Roy Want, Andy Hopper, Veronica Falcao, and Jonathan Gibbons. The active badge location system. *ACM Transactions on Information Systems*, 10(1):91–102, January 1992.
- [39] Patrick Widener, Fabian Bustamante, and Karsten Schwan. PDS - A proactive directory service. Technical report, Georgia Institute of Technology, December 2000.