Scalable Directory Services Using Proactivity

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

Abstract

Common to computational grids and pervasive computing is the need for an expressive, efficient, and scalable directory service that provides information about objects in the environment. We argue that a directory interface that 'pushes' information to clients about changes to objects can significantly improve scalability. This paper describes the design, implementation, and evaluation of the Proactive Directory Service (PDS). PDS' interface supports a customizable 'proactive' mode through which clients can subscribe to be notified about changes to their objects of interest. Clients can dynamically tune the detail and granularity of these notifications through filter functions instantiated at the server or at the object's owner, and by remotely tuning the functionality of those filters. We compare PDS' performance against off-the-shelf implementations of DNS and the Lightweight Directory Access Protocol. Our evaluation results confirm the expected performance advantages of this approach and demonstrate that customized notification through filter functions can reduce bandwidth utilization while improving the performance of both clients and directory servers.

1. Introduction

Innovative and widely distributed applications are enabled by infrastructure layers that allow distributed resources and services to be pooled and managed as though they were locally available. Advances in communication technologies and the proliferation of computing devices have made this possible; two such types of infrastructures are pervasive computing environments [11, 15] and computational grids [39, 16, 13]. An important component of both types of infrastructures is a directory service that provides information about different objects in the environment, such as resources and people, to applications and their users. Well-known examples of such services are the Metacomputing Directory Service (MDS) [12] for Globus-based environments, and the Intentional Naming System (INS) [1] for applications developed in the Oxygen [22] pervasive computing project. Directory services in both types of environments must support sophisticated object descriptions and query patterns, operate in highly dynamic environments, and scale to an increasingly large number of objects and users.

Traditional directory services have been designed for fairly static environments, where updates are rare (DNS [23], LDAP [43], and X.500 [29]). Recent work has addressed the issues of expressiveness of their object descriptions and query languages (through attribute-value hierarchies [1], for example) and considered their scalability (through domain-based partitioning or hierarchical organization [1, 8, 9, 33]). However, these directory services rely on traditional "inactive" interfaces, where clients interested in the values of certain objects' attributes must explicitly request such information from the server. Czajkowski et al. [8] demonstrate that it is feasible to satisfy widely different information service requirements with a single, consistent framework. Their example applications range from traditional service discovery with relatively static mappings to superschedulers and application adaptation monitors, where objects and their attributes change at fast and unpredictable rates and fresh information is crucial to clients' functionality. In such scenarios, clients in need of up-to-date information have no alternative but to query servers at rates that (at least) match those at which changes occur.

In this paper, we argue that an *exclusively* inactive interface to directory services can hinder server scalability and indirectly restrict the behavior of potential applications [32]. We propose to extend directory services' interfaces with a *proactive* mode by which clients can express their interest in, and be notified of, changes in the environment. A potential drawback of proactivity is the clients' loss of control of the frequency and type of notifications.

*0-7695-1524-X/02 $17.00 (c) 2002 IEEE

To address this, we propose the client-specific customization of notification channels through simple functions that are shipped and efficiently executed at the notifications' sources.

To validate our approach, we have designed and implemented the *Proactive Directory Service (PDS)* [4]. PDS is an efficient and scalable information repository with an interface that includes a proactive, push-based, access mode. Through this interface, PDS clients can learn of objects (or types of objects) inserted in/removed from their environment and about changes to pre-existing objects.

This work makes four contributions. First, we propose to extend the client interface to directory services with a proactive approach that allows clients to obtain up-to-date information at little or no extra cost in terms of client and server load.

Second, we propose the customization of notification on a per-client basis through the attachment of client-specific "filter" functions that are shipped to and efficiently executed at the notifications' sources.

Third, we present evaluation results that demonstrate the soundness of our implementation (by comparing its performance with that of off-the-shelf implementations of DNS and LDAP), and confirm the performance advantages that can be derived from proactivity in terms of network traffic and client and server load.

Finally, we demonstrate experimentally that, contrary to common wisdom, the customization of notification through filter functions executed at directory servers or objects' owners does not necessarily translate into overloaded servers. Indeed, this customization can improve the performance of notification sources as the cost of executing the additional filter code is outweighed by the gains resulting from eliminating unnecessary communications (i.e., executions of protocol stacks).

The remainder of this paper is structured as follows. Section 2 describes the ideas underlying our proactive approach to directory service interfaces. Section 3 provides a brief overview of the current prototype implementation of PDS. Section 4 discusses the performance and scalability benefits to be gained from proactivity and presents experimental results validating our hypothesis. We review related work in Section 5, and conclude in Section 6.

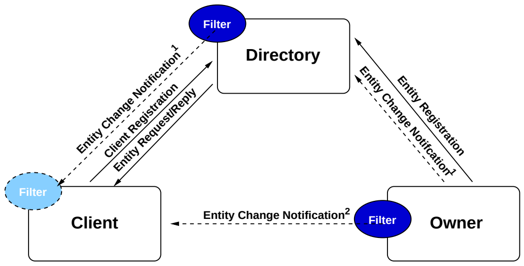

Figure 1. Owners register entities with the directory. Clients poll the directory for specific entities or types of entities. Clients can register to receive notifications of entity changes (from the directory (1) or the entity's owner (2)). Clients customize notification channels through filters placed at the notification sources.

2. Proactivity in Directory Services

Proactivity is a well-established system design technique with applications ranging from device/kernel communication to component-based software integration. The use of proactivity in directory services has some precedents in DNS NOTIFY [41] for zone change notification, the Ninja Secure Directory Service (SDS) [9] for service announcement, and the "persistent search" extension to LDAP proposed by Smith et al. [31].

Our proposal to extend directory services interfaces with proactivity has three parts: (1) we associate a channel for change notification with each object managed by the directory service, through these channels clients can become aware of changes to their objects of interest; (2) we support the customization of notification channels through client-specific filters, which are then used by the server to determine whether to send a given update; and (3) we adopt a leasing model for client registration to a notification channel that simplifies the handling of client failures. We now discuss each of these ideas in more detail.

Changes to an object managed by the directory service are reported to registered clients over the object's associated notification channel. Multiple clients can be registered with each notification channel and, conversely, a single client can be registered with multiple notification channels. Examples of types of events include the creation or removal of an entry or changes to (the attributes of) an existing entry in the directory.

An implicit attribute of passive, pull-based interfaces is *control*: clients are in control of the frequency and type of the messages exchanged with the directory service. Proactivity allows clients to trade control for performance, as message traffic is (only) generated when updates occur. After registration, however, clients are at the mercy of the service and can find themselves swamped with unforeseen (and potentially unwanted) updated messages.

At first glance, providing a filter at the client to discard the unwanted updates might seem enough. Although this does allow the application to ignore such updates, the corresponding 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 also 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 (in the form of a function) of "relevant" events. The server then uses these specifications, on a per-client basis, to determine whether to send a given update.

A critical issue then is the nature of the functions specifying clients' interest. Such functions could be expressed in a restricted filter language [6, 34] or in a general interpreted language such as Tcl/Tk [26] or Java [20]. A third approach, and the one adopted in PDS, is to allow specifications in a general (procedural) language, but to utilize dynamic code generation to create a native version of the functions at the notification source.

To avoid the unnecessary cost of pushing updates to clients who have failed (or terminated normally without unbinding) we advocate the use of leasing for registrations with notification channels. A lease, in this context, represents a period of time during which the request for change notification is active [1]. Clients can request a lease period, but the actual length of it is determined by the directory service. In addition, clients holding a lease can choose to cancel it or request its renewal.

The following section describes the architecture of our Proactive Directory Service and details its current prototype implementation. We present an overview of the language used for filter customization and describe the implementation of its dynamic code generation framework.

3. A Proactive Directory Service Prototype

To validate our approach we have designed and implemented PDS, a prototype of a proactive directory service. The PDS prototype is implemented in C/C++ on top of an event communication mechanism [10] and makes use of an efficient and extensible binary transport for communication in heterogeneous environments [2].

The PDS architecture includes three main components: PDS clients, servers and objects owners (Figure 1). PDS clients want to discover available objects in the environment and become aware of any change to them that could affect their functionality and/or performance. Objects owners make their objects available by publicizing them through the directory service. Servers act as mediators between clients and owners.

[1] Leasing also simplifies the handling of directory server failures, since such failures are perceived by clients of the service's proactive interface as lease cancellations.

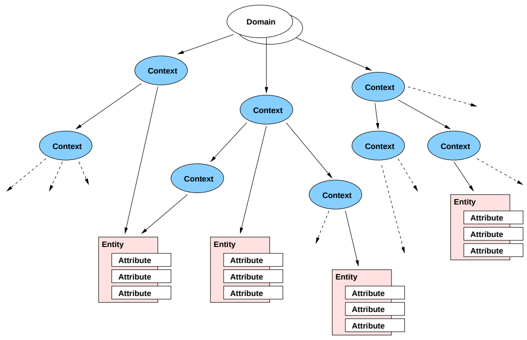

Figure 2. PDS data model: domains, contexts and entities. Each domain owns a root context in which all descendants are named.

As is common in directory services, related information in PDS is organized into well-defined collections called *entities*, where each entity represents an instance of an actual type of object in the environment (such as a host available for computation) and has an associated set of properties, or *attributes*, with particular values (such as CPU load or memory usage). Entities may be bound to names in different *contexts* and each context contains a list of name-to-entity bindings. In turn, contexts may themselves be bound to names in other contexts, building an arbitrary directed naming graph. Scalability of the global name space is obtained by dividing it into sub-spaces and assigns these sub-spaces to *domains*, each with a single *root* context.

Each object in PDS, be it a domain, a context, or an entity, has associated with it an event channel to which clients can subscribe. Each of the state-changing operations implemented by PDS submits an event to the notification channel associated with the appropriate object. When the owner of an object changes the value of some of the object's attributes (a significant drop on CPU availability at a given host, for instance), a notification is submitted to the associated channel.

The customization of these notification channels is done through client-specified filter functions written in ECL, a portable subset of C. These functions are shipped to the notification sources where there are dynamically compiled and installed. We now describe the capabilities of ECL and the current implementation of its translator.

3.1. ECL and Notification Channel Customization

The customization of notification channels, done on a per-client basis, is done through client-specified filter functions written in ECL. In these functions it is possible to examine the notification data and compare it with historical information to determine whether the notification should be

Table 1. ECL language basics. Types obey C promotion and conversion rules. Operators obey C precedence rules, and parentheses can be used for grouping.

Control-flow statements	
Statement	General form
for	`for` (*initial-statement*; *condition*; *iteration-statement*) *statement*
if-then-else	`if` (*condition*) *statement* `else` *statement*
return	`return` *statement*
Fundamental types	
Type class	Type
Integral types	`char`, `short`, `int`, `long`, `signed char`, `unsigned char`, `signed short`, `unsigned short`, `unsigned long`, `signed long`
Floating types	`float`, `double`
String types	`string`
Special return type	`void`
Basic operators	
Operator class	Operators
Arithmetic operators	+,-,*,/,%,
Boolean operators	!,&&,||,
Relational operators	⟩,⟨,⟨=,⟩=,==,!=
Assignment	=

sent (Figure 3).

ECL is a high-level language for dynamic code generation that targets specifically the generation of small code segments whose simplicity does not merit the cost and complexity of a large interpreted environment. ECL currently supports most C operators and fundamental data types, as well as static variables, function calls and basic control flow statements (Table 1). In terms of basic types, ECL does not currently support pointers, though the type `string` is introduced as a special case with limited support.

The implicit context in which filters are evaluated is a function of the form:

`int f(`⟨*notification type*⟩ `input)`

The return value of the function determines whether the notification would be issued to the registered clients.

In addition, filters can be parameterized by associating with them a set of read-only variables that clients can update remotely in a push-type operation. Through this, for example, a client can adjust the range of the filter shown in Figure 3.

ECL's dynamic code generation capabilities are based on Icode, an internal interface developed at MIT as part of the 'C project[28]. Icode supports dynamic code generation for MIPS, Alpha and Sparc processors and we have extended it to support MIPS n32 and 64-bit ABIs and x86 processors[2]. ECL consists primarily of a lexer, parser, semanticizer and code generator.

[2]Integer x86 support was developed at MIT. We extended Vcode to support the x86 floating-point instruction set (only when used with Icode).

```
{
  if ((input.cpuUsage < 0.1) ||
      (input.cpuUsage > 0.6)) {
    return 1;  /* submit event to channel */
  }
  return 0;  /* do not submit event */
}
```

Figure 3. An ECL filter that passes on notifications on changes to CPU utilization when the value is outside a specified range (in this case: $[0.1, 0.6]$**).**

4. Evaluation

In this section, we present evaluation results that confirm the expected performance advantages of a proactive approach. We first verify the fitness of our prototype implementation by comparing the performance of PDS support of the traditional pull-based interface with that of off-the-shelf implementations of DNS and LDAP. We use BIND DNS because it is a highly optimized directory system; OpenLDAP is an LDAP implementation that provides functionality and extensibility similar to PDS and forms the basis for MDS [12]. We then analyze the performance benefits of a proactive approach in the context of PDS.

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 updates. The costs of providing clients with up-to-date information (similar arguments apply to maintaining strong consistency in clients' caches [5]) can be measured in (1) client-service communication, (2) client CPU load and (3) server CPU load.

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.

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.

Given a randomly generated sequence of owner's updates and different desired degrees of consistency, we measure the load imposed on clients and servers as indicated by percentage of total CPU time consumed.

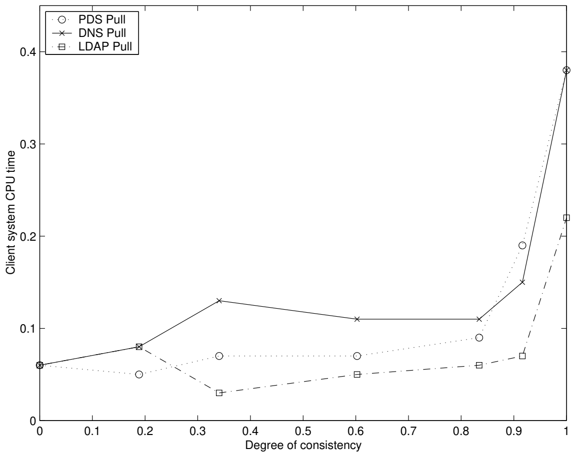

Figure 4. DNS, OpenLDAP and PDS client load (measured as percentage of total CPU time consumed) needed to reach a given degree of consistency through a traditional pull-based interface. Notice, however, that 100% consistency is inherently unachievable due to message latency.

4.1. Comparing DNS, LDAP and PDS Pull-Based Interface

We first verify the fitness of our prototype implementation by comparing the performance of PDS' pull interface against that of BIND DNS 8.2.2-7 [7] and OpenLDAP 1.2.1 [14]. For our experiments, all client-side caches were disabled and we ran DNS over TCP (by setting its 'vc' option) for purposes of comparison (OpenLDAP and PDS both use TCP for transport). Figure 4 and Figure 5 show the client and server loads for this three cases. We now analyze the performance benefits of a proactive approach in the context of PDS.

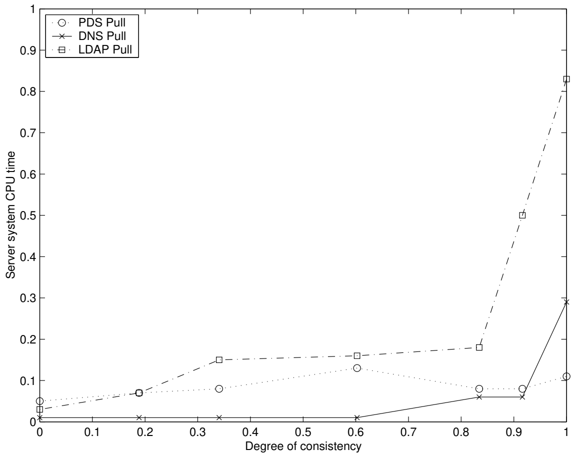

Figure 5. DNS, OpenLDAP and PDS server load (measured as percentage of total CPU time consumed) needed to reach a given degree of consistency through a traditional pull-based interface.

4.2. Performance Benefits from Proactivity

The benefits of a proactive interface for clients are clear from Figure 6. The figure shows that by using PDS's proactive mode, a (near) perfect (1) degree of consistency can be obtained, at a load on the client that is one-fourth that of under a pull-based interface. Note that advocate the extension, not the replacement, of traditional directory service interfaces with proactivity. Proactivity is beneficial when clients require up-to-date information but its benefits are less clear when lowers degrees of consistency are sufficient.

Figure 7 shows the benefits of proactivity to servers when clients require fresh information. It shows that through proactivity, a perfect degree of consistency can be obtained at a reasonably low server load. The scalability problems of a pull-based interface (as faced by PDS through its traditional interface, as well as the DNS and LDAP implementations) are clear.

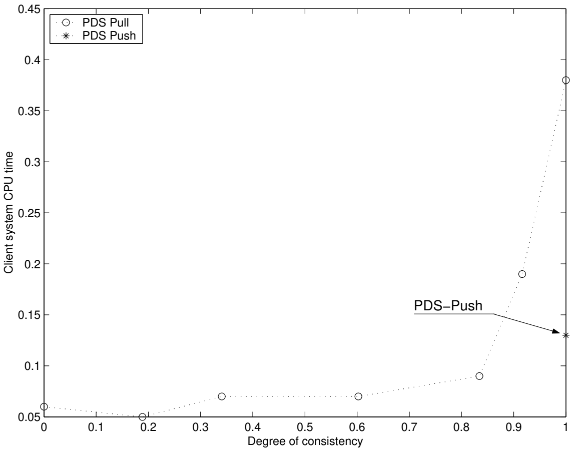

Figure 6. PDS Client load (measured as percentage of total CPU time consumed) needed to reach a given degree of consistency through PDS' traditional and proactive interfaces. Notice that the client load imposed by a proactive approach (PDS-Push) is shown as a single point since the experiment setup does not include filtering.

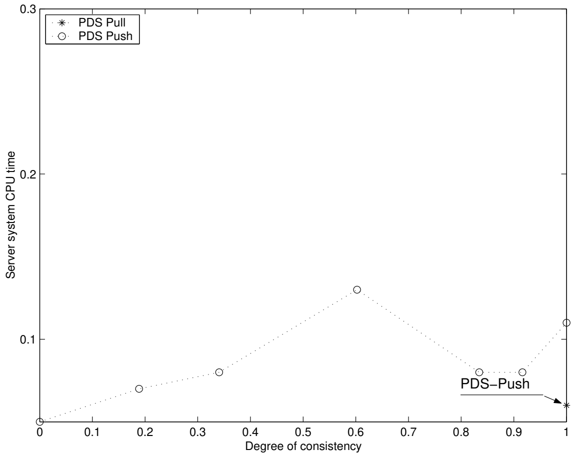

Figure 7. PDS Server load (measured as percentage of total CPU time consumed) needed to reach a given degree of consistency through PDS' traditional and proactive interfaces. Notice that the server load imposed by a proactive approach (PDS-Push) is shown as a single point since the experiment setup does not include filtering.

As previously mentioned, these experiments are only examples intended to illustrate the potential cost and benefits of our proactive approach. In fact, 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), this particular experiment imposes a load of only 400 requests/replies over 6 minutes (about 1.1 messages per second).

4.3. Evaluating the Costs of Client-Based Customization

In order to avoid the possible drawbacks of a proactive approach, we advocate the dynamic customization of notification channels through filter functions. The performance gains resulting from reduced network traffic come at the cost of generating and executing filter functions . Although these costs vary with the nature of the filter code, it is informative to examine some representative examples.

Table 2 compares the costs of code generation and execution of ECL with those of Java interpretation, the most likely alternative representation for filter functions. *Range* refers to the filter illustrated in Figure 3. The dynamic code generation for this filter requires 4 milliseconds on a Sun Sparc Ultra 30. The generated filter subroutine contains 27 Sparc instructions, and it executes in about 165 nanoseconds. In comparison, the same filter function implemented in Java requires 1.8 microseconds for execution with Just-In-Time compilation enabled (3.7 microseconds otherwise). A second filter, *array-average*, averages the entries of an incoming array of doubles before filtering out the notification. This filter is somewhat more complex that the previous one and the difference shows in the cost of dynamic code generation and, in particular, the cost of Java interpretation.

Initial experiments show that such customization, depending on the level of filtering, does not necessarily imply additional processing load, but can instead result in a net reduction (Figure 8). This is due to the fact that the additional cost of executing the filter code is out-weighed by gains in performance from the elimination of unnecessary message communications (i.e., executions of protocol stacks).

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

4.4. Server Scalability

As more clients register interest in entities maintained by PDS, the server must do more work in pushing changes when updates occur. As with any push-based solution, it is important that PDS performance scales well with increasing load. In this section we describe how increased load (in the

Table 2. Comparing ECL code generation and execution with Java interpretation.

Filter	ECL Generation	ECL Execution	Java JIT	Java
Range	4ms	165 ns	1.7 μs	3.7 μs
Array-average	5.5 ms	1.28 ms	13.33 ms	75.43 ms

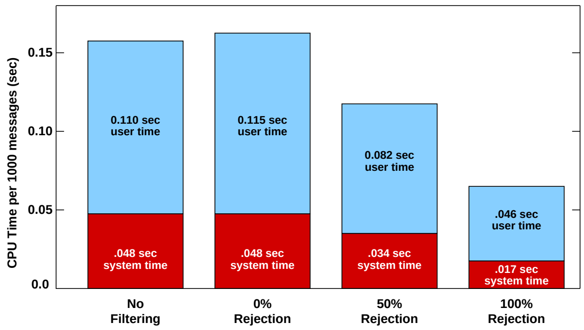

Figure 8. Server CPU utilization under different specialization scenarios.

form of an increased number of clients registered for updates) does not unreasonably degrade server performance.

Experimental setup. We constructed a set of experiments to test server scalability with a growing set of clients interested in updates. Our experiments were conducted on a cluster of 8-processor 550Mhz Intel Xeon machines, each having 4Gb of RAM connected via a dual gigabit Ethernet network backplane. We allocated a maximum of 5 clients on any one machine, so that each client was guaranteed to have a processor available. Additionally, the clients were started in an interleaved fashion across machines.

Each experiment consists of a set of 1000 event pushes to the set of interested clients. This set varies in size between 1, 10, 20, 30, 40, and 50 clients. We used a single event size of roughly 100 bytes. This size is comparable to a majority of update events generated by PDS. We believe this to be a reasonable decision given that we expect directory services to steward small pieces of data and so it is efficient to deliver updated data along with the update notification. PDS can be configured to deliver only notifications if necessary. Also, larger event sizes should not affect the scalability of the server in the face of increasing numbers of clients; they do not contribute to overhead at the server and represent a proportional increase in work that is unrelated to the number of clients served.

For each experimental scenario, we report several metrics. Total send time is measured as the "wall-clock" time needed by the server to fully distribute the set of updates

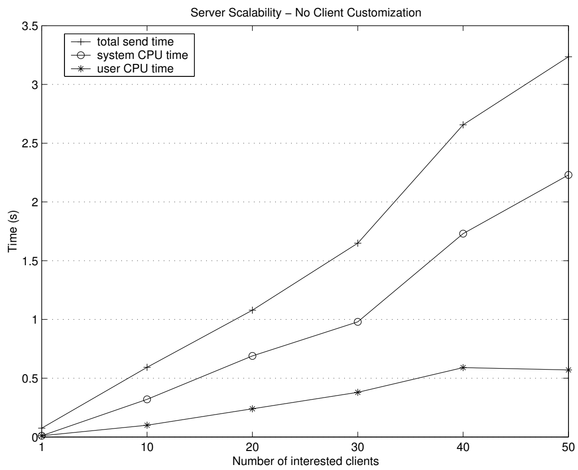

Figure 9. Server scalability with no client customization.

among the set of clients. As the underlying event channel middleware we use, ECho, is based on TCP, an event send involves a TCP `write()` to each client. Real-time measurements in this section begin with the first event send and terminate at the end of the last server-side `write()`. We measure a large enough sample size of events that TCP buffering effects are negligible. Server load is expressed as the amount of user and system time elapsed in distributing the updates.

We now describe the results of our experiments. We tested scenarios in which clients do not customize the event channel as well as those where clients discard varying proportions of the event stream. We also tested cases in which the proportions of the event stream were not the same across the entire set of clients.

No client customization. In the basic case, clients register interest in updates and receive every one without any customization or filtering. As the number of clients increases, the server must distribute all updates to all interested clients.

Figure 9 shows that our basic measures (real-time, user-state CPU time, and system-state CPU time) increase roughly linearly with the number of clients. This is to be expected given the one-to-one nature of the underlying trans-

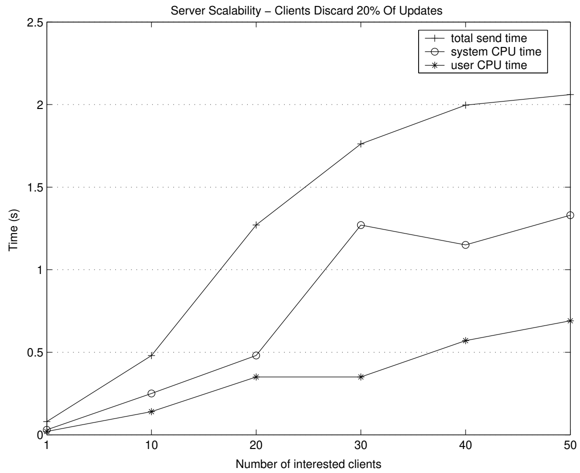

Figure 10. Server scalability where clients discard 20% of update traffic.

port.

Clients using filters. As described previously, a key benefit of the proactive mode implemented by PDS is the ability of clients to throttle the rate of updates through customization of the event channel. Figures 10, 11, and 12 illustrate scenarios where the set of clients discard 20%, 50%, and 80% of updates, respectively.

Figure 10 shows a slight increase in user CPU time compared to the no-filter case. This is due to the execution of the filter that decides whether or not to discard the update. At the higher discard rates shown in Figures 11 and 12, the user CPU time drops, reflecting the savings realized by not executing the TCP stack for discarded updates.

These results show that the measured server load increases in a linear manner. However, it is also important to note that the absolute time required to send the entire set of updates decreases greatly with increasing customization of the event channel. This very clearly illustrates the power of combining proactive notification with client control of update rate.

Although it is reasonable to assume that most clients of a directory service will use similar customizations (discard updates when the update rate exceeds a certain value, or based on a data item in the update itself), it would be questionable to assume that all clients would use the same discard criteria. Accordingly, Figures 13 and 14 depict instances where two different proportions of the same update stream are being discarded. In both cases half of the clients discard 80% of update traffic, but the other half discards 20% and 50%, respectively.

Although not as favorable as the single-customization

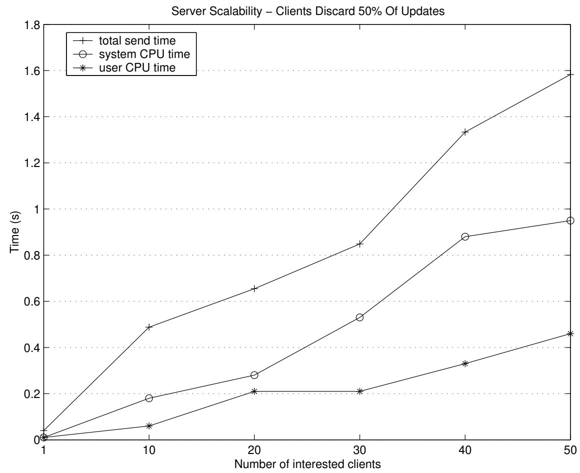

Figure 11. Server scalability where clients discard 50% of update traffic.

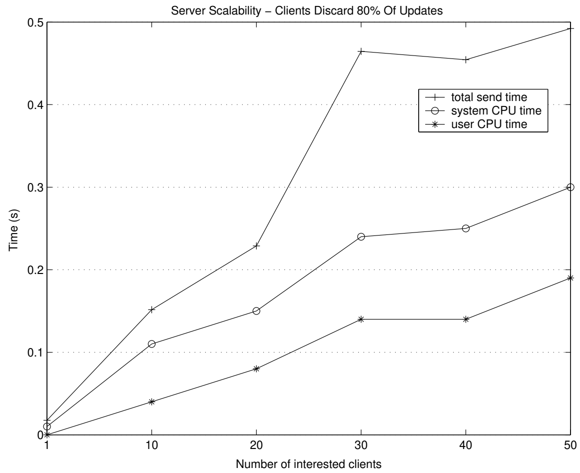

Figure 12. Server scalability where clients discard 80% of update traffic.

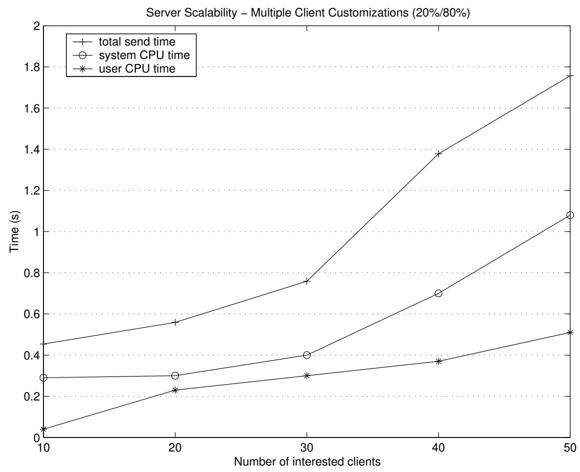

Figure 13. Server scalability where half of the clients discard 20% of update traffic and half discard 80%.

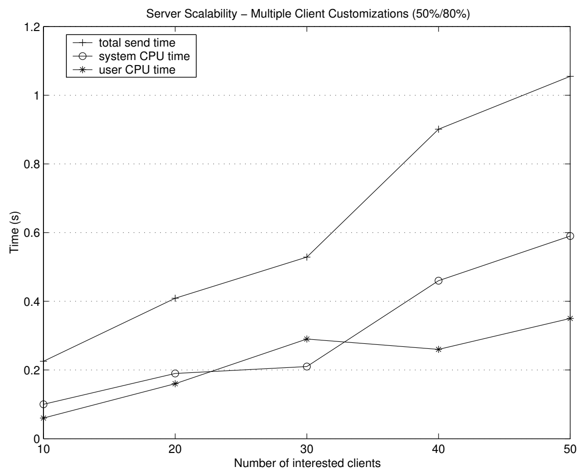

Figure 14. Server scalability where half of the clients discard 50% of update traffic and half discard 80%.

cases, these results also indicate that a push-update system can scale adequately as the number of client connections increases. The absolute reduction in CPU times also holds in the 50%/80% case, further demonstrating the scalability of the server.

Middleware enhancements. In the final analysis, the scalability of PDS or any other push-update mechanism is bounded by the capabilities of the underlying event channel middleware. Our group is actively working to develop more scalable middleware for multicast/broadcast situations. Specifically, we are currently modifying our transport layer to use IP-Multicast. Reliability and further adaptability will be guaranteed by an adaptive Reliable UDP package we have developed [19]. When these changes are in place, clients with interest in the same data items will occupy the same multicast group, allowing true event broadcast instead of a sequence of unicasts. We are also addressing wide-area event distribution through the use of an overlay network to efficiently route update traffic.

5. Related work

There are many variants on the common theme of directory services. Classical directory services [23, 33, 29] were designed under the assumption of fairly stable mappings between objects' attributes and their values.

Active Names [38] and INS [1] concentrate on the problem of efficient and flexible name-to-object resolution, as opposed to PDS' emphasis on providing low-impact up-to-date information to clients. 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.

Directory services supporting entries with attributes have made feasible service and resource discovery by processing queries containing a set of desired attributes [42, 27, 40, 9]. PDS also supports retrieval of entities based on attribute-value pairs. While Jini relies on Java RMI as a transport mechanism; PDS uses a fast binary transport encoding mechanism that provides superior performance [2]. PDS does not currently address issues of authentication and secure communication as does SDS.

Czajkowski et al. [8] present the architecture of MDS-2, the new generation of MDS [12]. Their framework is intended to satisfy widely different information service requirements. Their infrastructure currently supports only a traditional pull-based client interface, although the incorporation of a proactive extension is part of their planned future work.

The use of proactivity in directory services has some precedents [41, 9]. All these proposals, however, maintain a passive client interface and, thus, their associated scalability problems.

Proactivity is perhaps most closely related to *persistent searches* as proposed by Smith et al. [31]. Persistent searches are an extension to LDAP that would allow clients to receive notification of changes in a server by allowing standard search operations to remain active until abandoned by the client (or until the client unbinds). In addition, our approach considers the dynamic customization of notification channels through client-specified functions to recover some of the client's control inherent in traditional pull-based interfaces.

The Global Grid Forum's Performance Working Group is developing a Grid Monitoring Architecture (GMA) [37] for computational grids. GMA is specifically aimed at performance-monitoring and could thus benefits from the characteristics of this type of information (such as short lifetime, frequent update rate and stochastic nature). Although PDS is a general directory service, it shares a number of architectural ideas with GMA including their three main architectural components, the components' roles and their types of interaction. We believe PDS could potentially be used to implement an "extended" GMA and we have started to explore this research path with *AIMS*, an *Adaptive Introspective Management System* [3].

Our work on PDS has some similarities with research on wide-area cache consistency [18, 5]. We affirm the conclusions presented in both papers about the impact of client polling on server load and network usage. When relying on invalidation, clients are still responsible for retrieving the updated web page from the server. PDS has the ability to supply the updated object/attribute data in the update message, thereby saving a message exchange; in addition, it provides clients with the possibility of customizing such updates to their own needs.

We propose the extension of directory services' interfaces with a customizable push-based mode. There is a significant body of related work in push-based delivery systems and publish-subscribe infrastructures as well as some standardization efforts. Yeast [21] is an event-action system with a rich event pattern language where actions are specified as UNIX shell scripts. AT&T Labs' READY [17] extends Yeast with a set of high-level constructs and a richer specification language that allows compound matching events and QoS directives. Elvin [30] is a centralized event dispatcher with an expressive event-filtering language. Siena [6] and Gryphon [34] are both content-based message-brokering systems while our group's ECho [10] and the Java Distributed Event Specification [35] adopt a channel-based addressing scheme. Yu et. al [44] proposed an event notification service with a novel peer-to-peer architecture of proxy servers. The CORBA Event Service [25] and the Notification Service [24] specification, as well as the Java Message Service [36] are well-known efforts to specify event notification services. PDS relies on ECho to implement the client-customizable channels that associates with each of its objects.

6. Conclusion and Future Work

We have presented a case for the support of proactive interfaces by directory services based on their potential benefit to service scalability. To address the possible drawbacks of a push-based approach, we have proposed the use of client-specific server customization. In order to validate our ideas, we have designed and built PDS, a proactive directory service that implements our approach. Our experimental results show that (i) directory services with exclusively passive interfaces can encounter scalability problems when used in new types of environments [11, 39, 16, 13] with large numbers of objects and highly dynamic mappings [8], and that (ii) a customizable form of proactivity can avoid these problems without sacrificing control.

Some very important areas of research remain, including the incorporation of security mechanisms and extensions to the language used for customization. In addition, we are working on the use of proactivity to enhance the robustness of widely distributed services through flexible replication strategies, dynamically adaptive server hierarchy management, adaptive introspection and automatic failure recovery.

Acknowledgments

We are grateful to Greg Eisenhauer for his assistance with profiling the scalability of the ECho middleware package.

References

[1] W. Adjie-Winoto, E. Schwartz, H. Balakrishnan, and J. Lilley. The design and implementation of an intentional naming system. In *Proc. of the 17th ACM Symposium on Operating System Principles*, pages 186–201, Kiawah Island, SC, December 1999. ACM.

[2] F. E. Bustamante, G. Eisenhauer, K. Schwan, and P. Widener. Efficient wire formats for high performance computing. In *Proc. of Supercomputing 2000 (SC 2000)*, Dallas, TX, November 2000.

[3] F. E. Bustamante, C. Poellabauer, and K. Schwan. Aims: Robustness through sensible introspection. In *Proc. of the Tenth ACM SIGOPS European Workshop*, Saint-Emilion, France, September 2002.

[4] F. E. Bustamante, P. Widener, and K. Schwan. The case for proactive directory services. In *Proc. of Supercomputing 2001 (SC 2001)- Poster Session*, Denver, CO, November 2001.

[5] P. Cao and C. 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.

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

[7] I. S. Consortium. Bind domain name service. http://www.isc.org/products/BIND/bind8.html.

[8] K. Czajkowski, S. Fitzegerald, I. Foster, and C. Kesselman. Grid information services for distributed resource sharing. In *Proc. of the 10th High Performance Distributed Computing (HPDC-10)*, San Francisco, CA, August 2001.

[9] S. Czerwinski, B. Zhao, T. Hodes, A. Joseph, and R. Katz. An architecture for a secure service discovery service. In *Proc. of ACM/IEEE MOBICOM*, pages 24–35, August 1999.

[10] G. Eisenhauer, F. E. Bustamante, and K. Schwan. Event services for high performance computing. In *Proc. of the 9th High Performance Distributed Computing (HPDC-9)*, Pittsburgh, PA, August 2000. IEEE.

[11] M. Esler, J. Hightower, T. Anderson, and G. Borriello. Next century challenges: Data-centric networking for invisible computing. In *Proc. of the 5th ACM/IEEE International Conference on Mobile Computing and Networking*, pages 24–35, Seattle, WA, August 1999.

[12] S. Fitzgerald, I. Foster, C. Kesselman, G. von Laszewski, W. Smith, and S. Tuecke. A directory service for configuring high-performance distributed computation. In *Proc. of the 6th High Performance Distributed Computing (HPDC-6)*, pages 365–375, Portland, OR, August 1997. IEEE.

[13] I. Foster and C. Kesselman. Globus: A metacomputing infrastructure toolkit. *International Journal of Supercomputing Applications*, 11(2):115–128, 1997.

[14] O. Foundation. The OpenLDAP Project. http://www.openldap.org//

[15] R. Grimm, T. Anderson, B. Bershad, and D. Wetherall. A system architecture for pervasive computing. In *Proc. of the 9th ACM SIGOPS European Workshop*, pages 177–182, Kolding, Denmark, September 2000.

[16] A. Grimshaw, A. Ferrari, F. Knabe, and M. Humphrey. Legion: An operating system for wide-area computing. *IEEE Computer*, 32(5):29–37, May 1999.

[17] R. E. Gruber, B. Krishnamurthy, and E. Panagos. High-level cosntructs in the READY event notification service. In *Proc. 8th ACM SIGOPS European Workshop on Support for Composing Distributed Applications*, Sintra, Portugal, September 1998.

[18] J. Gwertzman and M. Seltzer. World-Wide Web cache consistency. In *Proc. of the 1996 USENIX Technical Conference*, January 1996.

[19] Q. He and K. Schwan. IQ-RUDP: Coordinating application adaptation with network transport. In *Proc. of the 11th High Performance Distributed Computing (HPDC-11)*, pages 369–378, Edinburgh, Scotland, July 2002.

[20] D. Kramer. The Java platform: A white paper. *Sun Microsystems Inc*, May 1996.

[21] B. Krishnamurthy and D. S. Rosenblaum. Yeast: a general purpose event-action system. *IEEE Transactions on Software Engineering*, 21(10):845–857, October 1995.

[22] Massachusetts Institute of Technology. MIT project Oxygen. http://www.lcs.mit.edu//

[23] P. Mockapetris and K. J. Dunlap. Development of the domain name system. In *Symposium proceedings on Communications architectures and protocols (SIGCOM'98)*, pages 123–133, Stanford, CA, August 1988. ACM.

[24] OMG. Notification service specification 1.0. ftp://www.omg.org/pub/doc/formal//0-06-20.pdf, June 2000.

[25] OMG. Event service specification 1.1. ftp://www.omg.org/pub/docs/formal//1-03-01.pdf, March 2001.

[26] J. K. Ousterhout. *Tcl and the Tk Toolkit*. Addison-Wesley, 1994.

[27] C. Perkins. Service location protocol white paper. http://playground.sun.com/srvloc/slp_white_paper.html, May 1997.

[28] M. Poletto, D. Engler, and M. F. Kaashoek. tcc: A template-based compiler for 'c. In *Proc. of the First Workshop on Compiler Support for Systems Software (WCSSS)*, February 1996.

[29] S. Radicati. X.500 directory services: Technology and deployment. Technical report, International Thomson Computer Press, London, UK, 1994.

[30] B. Segall and D. Arnold. Elvin has left the building: a publish/subscribe notification service with quenching. In *Proc. of AUUG97*, pages 243–255, Brisbane, Australia, September 1997.

[31] M. Smith, G. Good, R. Weltman, and T. Howes. Persistent search: a simple LDAP change notification mechanism. Working document, IETF, November 2000. draft-smith-psearch-ldap-00.txt.

[32] W. Smith, A. Waheed, D. Meyers, and J. Yan. An evaluation of alternative designs for a grid information service. In *Proc. of the 9th High Performance Distributed Computing (HPDC-9)*, Pittsburgh, PA, August 2000. IEEE.

[33] M. V. Steen, F. J. Hauck, and A. S. Tanenbaum. Locating objects in wide-area systems. *IEEE Communication Magazine*, pages 104–109, January 1998.

[34] R. Strom, G. Banavar, T. Chandra, M. Kaplan, K. Miller, B. Mukherjee, D. Sturman, and M. Ward. Gryphon: An information flow based approach to message brokering. In *International Symposium on Software Reliability Engineering '98 Fast Abstract*, 1998.

[35] Sun Microsystems. Java distributed event specification. Moutain View, CA, 1998.

[36] Sun Microsystems. Java message service. Moutain View, CA, 1999.

[37] B. Tierney, R. Aydt, D. Gunter, W. Smith, V. Taylor, R. Wolski, and M. Swany. A grid monitoring architecture. Grid Working Draft GWD-Perf-16-2, Global Grid Forum – Performance Working Group, January 2002.

[38] A. Vahdat, M. Dahlin, T. Anderson, and A. Aggarwal. Active names: flexible location and transport of wide-area resources. In *Proc. of USENIX Symp. on Internet Technology & Systems*, October 1999.

[39] M. van Steen, P. Homburg, , and A. Tanenbaum. Globe: A wide-area distributed system. *IEEE Concurrency*, 7(1):70–78, January-March 1999.

[40] J. Veizades, E. Guttman, C. Perkins, and S. Kaplan. Service location protocol. RFC 2165, Network Working Group, June 1997.

[41] P. Vixie. A mechanism for prompt notification of zone changes (dns notify). RFC 1996, Network Working Group, August 1996.

[42] J. Waldo. The Jini architecture for network-centric computing. *Communication of the ACM*, 42(7):76–82, July 1999.

[43] W. Yeong. Lightweight directory access protocol. RFC 1777, Network Working Group, March 1995.

[44] H. Yu, D. Estrin, and R. Govindan. A hierarchical proxy architecture for internet-scale event services. In *Proc. of WET-ICE'99*, Stanford, CA, June 1999.