

Dynamic Differential Data Protection for High-Performance and Pervasive Applications

A Thesis
Presented to
The Academic Faculty

by

Patrick M. Widener

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy

College of Computing
Georgia Institute of Technology
August 2005

Copyright © 2005 by Patrick M. Widener

Dynamic Differential Data Protection for High-Performance and Pervasive Applications

Approved by:

Karsten Schwan, Ph.D., Advisor
College of Computing
Georgia Institute of Technology

Calton Pu, Ph.D.
College of Computing
Georgia Institute of Technology

Mustaque Ahamad, Ph.D.
College of Computing
Georgia Institute of Technology

Andrew Grimshaw, Ph.D.
Department of Computer Science
University of Virginia

Douglas Blough, Ph.D.
School of Electrical and
Computer Engineering
Georgia Institute of Technology

Date Approved: 18 July 2005

What a long, strange trip it's been.

— J. G.

The point of a journey is not to arrive.

— N. P.

“ “

— B. M. G.

in memory of all i abandoned

in celebration of all i did not

ACKNOWLEDGEMENTS

While this work would obviously not have happened without my time and effort, it also would not have been possible without that of many other people. I have been fortunate to have had their advice, support, and companionship.

I arrived at Georgia Tech a competent technologist, but I will depart a competent academic researcher. The lion's share of credit for that non-trivial transformation belongs to my advisor, Karsten Schwan. There is, quite simply, no way to fully express my gratitude for his guidance, consideration, and encouragement. My confidence in myself wavered, but his confidence in me never did.

The members of my thesis committee have been patient and supportive as I've completed this dissertation. Mustaque Ahamad has graciously served as a sounding board for my ideas, cheerfully giving me feedback on my research while tolerating its idiosyncracies. Calton Pu and Doug Blough have each provided unique perspectives on my research and challenged me to present it in the best possible way. Andrew Grimshaw made time out of a notoriously busy schedule to be a part of my thesis committee, and his commentary has been extremely valuable.

My experiences, personal and professional, with fellow students and collaborators from the College of Computing have been the unexpected reward of my graduate school experience. Among them, Fabian Bustamante, Greg Eisenhauer, Vernard Martin, Ada Gavrilovska, Matt Wolf, Joe Tullio, Idris Hsi, and Josh Fryman have helped me to cope, to laugh, and to be a better student, researcher, and person.

Finally, it is not overstatement or melodrama to say that my close friends in most cases kept me sane, in some cases kept me fed, and in all cases gave me reasons to keep going when I had few. That they are not all named here speaks only to the present limitations of space and time; in my head and heart, those limits do not apply. Andy, Jenn, Robynn, Fran, Angie, Melissa, Sally, Jennifer, Dave, Kevin, and all the rest - thank you.

TABLE OF CONTENTS

ACKNOWLEDGEMENTS	iv
LIST OF FIGURES	viii
SUMMARY	x
CHAPTER 1 INTRODUCTION	1
1.1 A Distributed Application With Limited Functionality	1
1.2 Dynamic Differential Data Protection	3
1.3 Motivations: Data Exchange in Distributed Applications	4
1.4 Threat Analysis	6
1.5 Thesis Statement And Dissertation Organization	9
CHAPTER 2 APPLICATION DOMAINS	11
2.1 Pervasive Applications: Surveillance	14
2.2 High Performance Computing: Data-centric Collaboration	17
2.3 Enterprise Applications	19
CHAPTER 3 AN IMPLEMENTATION OF D3P IN PUBLISH/SUBSCRIBE MIDDLEWARE	23
3.1 ECho Fundamentals	23
3.2 Overview Of Basic ECho/D3P Usage	25
3.3 Implementing Protected Objects	26
3.3.1 Protected References to ECho Objects	26
3.3.2 ECho Object Modifications	28
3.4 Implementing Per-Client Customization	29
3.4.1 Customizing Event Delivery In ECho	30
3.4.2 Using ECho Derived Channels	32
3.4.3 Providing Efficient Data Protection Through Type Manipulation	33
3.4.4 <i>a priori</i> Channel Customization	34
3.5 Overwatch, Application Metadata, And Policy Management	34
3.5.1 Overwatch Basics	35
3.5.2 Metadata Used by Overwatch	36

3.6	Toward a Model for D3P Application Design	38
3.6.1	Model Objects	38
3.6.2	Capabilities	42
3.6.3	Type Definitions	43
3.6.4	Degrees of Freedom	44
3.7	Model vs. Implementation	44
CHAPTER 4 INFRASTRUCTURE COMPONENTS FOR DATA PROTECTION		47
4.1	Data Type Services	47
4.1.1	PBIO	47
4.1.2	XMIT	48
4.2	Directory Services	51
4.2.1	Proactivity in Directory Services	52
4.2.2	Proactive Directory Service in a D3P Infrastructure	54
4.2.3	Broker Services	55
4.3	Summary	55
CHAPTER 5 SUPPORTING AUDIT AND CONFIGURATION CONTROL USING D3P		57
5.1	Reverb Background	58
5.2	The Reverb Subsystem	61
5.2.1	Reverb Mechanisms	61
5.2.2	Applying Reverb With Checked Access	62
5.2.3	Customizing the RChannel	63
5.2.4	Supporting Infrastructure	64
5.3	Application Experience	64
5.3.1	Active Video Streams	65
5.3.2	Configuration Change in AVS	66
5.3.3	Access to the RChannel	67
5.3.4	Customization of the RChannel	68
5.3.5	Response to Changes	69
5.4	Reverb Summary	70

CHAPTER 6	EVALUATION	72
6.1	Data Path vs. Control Path	73
6.2	Basic D3P Overhead	73
6.3	Scalability	74
6.3.1	Reverb Measurements	76
6.3.2	Customization	78
6.4	XMIT Evaluation	78
6.4.1	Impact on Application Size	84
6.4.2	XMIT Summary	84
6.5	Performance Improvement	84
6.6	Evaluation Summary	87
CHAPTER 7	RELATED WORK	88
7.1	Trusted Computing	88
7.2	User-level Protection Mechanisms	88
7.3	Security Policy Implementation and Management	90
7.4	Protected Audit Systems	92
7.5	Middleware	94
7.5.1	Component-Based Systems	94
7.5.2	Publish/Subscribe	94
7.6	Active and Adaptive Systems	95
CHAPTER 8	CONCLUDING REMARKS	98
8.1	Summary	98
8.2	Future Work	99
REFERENCES		101
VITA		115

LIST OF FIGURES

Figure 1	A remote-sensing application.	1
Figure 2	The AVS (Active Video Streams) application.	14
Figure 3	Command Center Interface.	15
Figure 4	C language structure representing a PPM image used by the AVS application.	15
Figure 5	The SmartPointer application. Scientific users can share information about the molecular dynamics simulations on heterogeneous output devices ranging from wirelessly-connected handhelds to those with advanced 3-dimensional rendering capabilities.	17
Figure 6	Architecture of the Operational Information Systems mid-tier.	20
Figure 7	Processes using Event Channels for Communication.	24
Figure 8	Derived event channels are created by applying a function to the output of an existing event channel.	30
Figure 9	Installing a handler in AVS requires the use of several D3P objects. An <i>EControlContext</i> (1) is needed in order to create any other protected ECho objects. Capabilities for the original, unmodified <i>EChannel</i> (2), the P BIO types involved (the type used in the original channel and the one to be generated by the newly installed handler, in an P BIO <i>IOContext</i>) (3), and for the handler itself (4) are required for the handler installation. Communication with Overwatch (5) is necessary to retrieve at least the handler capability. The result of the operation (6) is a new protected <i>EChannel</i> object (7) which can then be used by AVS.	32
Figure 10	Use of metadata in the AVS application.	36
Figure 11	Illustration of the D3P system model.	39
Figure 12	Inspectors, Adorners, and Morphers take different actions on input and output types.	40
Figure 13	Handlers can be combined in <i>graphs</i> to achieve a series of effects.	42
Figure 14	Capabilities in D3P can be viewed as having record structure.	43
Figure 15	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.	53

Figure 16	The Active Video Streams (AVS) applications consists of a camera driven by a separate host machine. This host machine serves as a video source and transmits images from the camera over a wireless communication link to a Java player application. The player application incorporates a control interface which can install filters on the event channel connecting the two hosts.	65
Figure 17	ECho scalability with no D3P features.	75
Figure 18	ECho scalability where increasing numbers of clients use the D3P interfaces.	76
Figure 19	ECho scalability under increasing numbers of clients with a fixed set of Reverb subscribers.	77
Figure 20	Server scalability where half of the clients discard 20% of Reverb traffic and half discard 80%.	79
Figure 21	Server scalability where half of the clients discard 50% of Reverb traffic and half discard 80%.	80
Figure 22	A C structure defining a sample structure representing an image, and a sample XML encoding of the structure. The XML expansion results in a considerably larger representation of the data, significant when exchanging many messages.	81
Figure 23	Format registration costs using PBIO and XMIT.	83
Figure 24	ECho/D3P scalability as event sizes increase.	85

SUMMARY

Modern distributed applications are long-lived, are expected to provide flexible and adaptive data services, and must meet the functionality and scalability challenges posed by dynamically changing user communities in heterogeneous execution environments. The practical implications of these requirements are that reconfiguration and upgrades are increasingly necessary, but opportunities to perform such tasks offline are greatly reduced. Developers are responding to this situation by dynamically extending or adjusting application functionality and by tuning application performance, a typical method being the incorporation of client- or context-specific code into applications' execution loops.

Prior work has highlighted the performance advantages provided by dynamic code extension or specialization. Our work addresses a basic roadblock in deploying such solutions, which is the protection of key application components and sensitive data in distributed applications. Our approach, termed Dynamic Differential Data Protection (D3P), provides fine-grain methods for providing component-based protection in distributed applications. Context-sensitive, application-specific security methods are deployed at runtime to enforce restrictions in data access and manipulation. D3P is suitable for use in low- or zero-downtime environments, since such deployments are performed while applications run. D3P is appropriate for high performance environments and for highly scalable applications like publish/subscribe, because it creates native codes via dynamic binary code generation. Finally, due to its integration into middleware, D3P can run across a wide variety of operating system and machine platforms.

This dissertation introduces the need for D3P, using sample applications from the high performance and pervasive computing domains to illustrate the problems addressed by our D3P solution. It also describes how D3P can be integrated into modern middleware. We present experimental evaluations which demonstrate the fine-grain nature of D3P, that is, its ability to capture individual end users' or components' needs for data protection, and

also describe the performance implications of using D3P in data-intensive applications.

CHAPTER 1

INTRODUCTION

This dissertation proposes and evaluates data management methods for applications in the high-performance and scientific domains, providing functionality and overcoming issues not addressed by traditional techniques. For applications in these domains, where data is generated from and distributed to heterogeneous and possibly anonymous endpoints, our results demonstrate that non-trivial data differentiation and protection are possible with small impact on application performance. We also show that in a significant set of such applications, our methods improve performance over alternative solutions providing equivalent functionality. Furthermore, we demonstrate the practicality of implementing our solution by relying on widely-accepted data specification standards and embedding it into a mature, high-performance data transport infrastructure.

1.1 A Distributed Application With Limited Functionality

Consider a distributed application (as in Figure 1) that transmits data from a remotely-located sensor to an end-user application. Without using a middleware package, the developer of this application is forced to write all the communication code himself. In addition, that communications code must be now tightly bound to the “domain” code, and changes in one will cause either known effects in the other which must be addressed or unknown effects which may cause faults during later development or deployment.

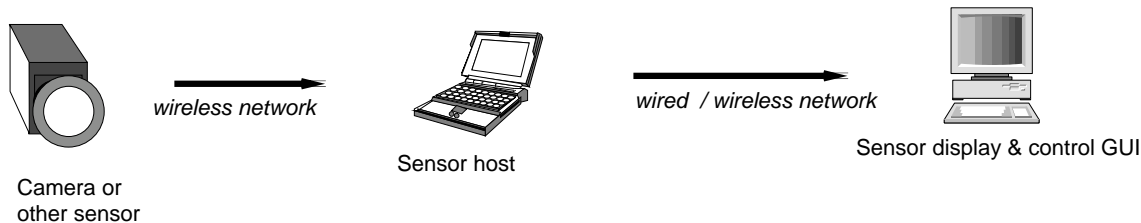


Figure 1: A remote-sensing application.

Simply saying “use middleware” for this problem is not a solution, however. Even if basic problems such as abstracting communications code and wire-formatting are addressed, several issues remain:

- How will per-user customization be handled? Many user changes to data are small and relatively easy to implement (cropping an image, for example). However, pushing such changes into the middleware itself dilutes the abstractions provided by the middleware. More importantly, it forces a middleware software release cycle to occur before user needs are satisfied, a huge barrier to user acceptance.
- How is the designer of the application to accommodate device types unknown at design-time? A sensor application that transmits images could potentially have to send 33% more data in order to accommodate both color and greyscale displays, for example.
- How will applications cope with application policies unknown at design time? If actions dictated by policy (such as “Alice cannot see the northwest quadrant of the image when the camera orientation is between north and northeast”) are usefully implemented in middleware for performance and encapsulation reasons, how will middleware designers address these issues? What if, in the freight yard Alice is observing, the freight car Alice is restricted from seeing moves? This invalidates the old policy and presumably drives the creation of a new one (“*northeast* quadrant of the image”). How will the new policy be implemented? And in general, how can policy makers express restrictions in a way enforceable by middleware?

Existing middleware addresses many of these issues, but none in a comprehensive way. Many middlewares support the incorporation of user codes, but the protection of the data on which those codes operate is ignored. Middlewares that do provide implementation of externally-defined security policies frequently do not provide ways to associate policy information with metadata. Middleware architectures that do allow such associations (incorporating relational or other database mechanisms) do not scale to the data sizes, data

distribution requirements, endpoint diversity or user populations required by modern applications. Other middleware security approaches concern themselves with policy definition and management, authentication schemes (such as group authentication [166] or distributed single-sign-on [111, 21, 46]), or auditing [88]. A comprehensive approach to data management for the middleware used in streaming-data applications has heretofore been unavailable.

1.2 Dynamic Differential Data Protection

In comparison to such research, this dissertation focuses on more general solutions for data management and explores how they can be leveraged to provide data protection. We pose problems and devise solutions for:

1. *Information access: typed data.* How is access to remote information governed? Stated more specifically, what access or protection model governs a remote user's ability to access some information items and not others? Our solution approach uses the middleware's type system, that is, its support for typed data, in order to support stateful data inspection and to differentiate across multiple agents' accesses to the same data items.
2. *Authorization: capability-based model.* Once remote users have been authenticated and have established their ability to access certain information, how to express and implement restrictions concerning information access? Our solution uses a capability-based model, where capabilities are associated with data streams, the data types those streams carry, and user code designed to customize them.
3. *Extension and adaptation: runtime code generation and deployment.* What mechanisms enforce restrictions on data access? Our solution is to use dynamically generated, safe code to implement the flexible restrictions needed by today's applications, and then use capabilities to express permissions and enforce restrictions on dynamic code deployment.

4. *Locus of control: middleware- vs. system-level support.* Rather than relying on specialized operating systems, our approach uses a set of middleware abstractions to implement fine-grain restrictions on data access. The outcome is improved flexibility and extensibility compared to system-level solutions, examples including the ability to incorporate application-level requirements and security policies via runtime parameterization or the expression of utility functions.

The fundamental concept underlying (1)-(4) is that of *dynamic, differential data protection* (D3P).

D3P provides *differential* support for protecting application data and/or components. Data differentiation uses metadata about the application-level information being accessed to adjust the granularity of access control. A single access control decision can be applied to a single user or an entire application with equal ease. Differential access to data streams produces customized, policy-driven access to data, without duplicating any stream content.

D3P leverages data differentiation to provide *protection* for data. A basic data access model derived from prior work on object- and capability-based models of data protection [109] is used. In D3P, typed data is accessed and manipulated only by well-defined operations, and applications are allowed access to exactly the types and operations specified by policy.

To accommodate modern distributed applications, D3P permits the code fragments that operate on data objects to be generated and deployed *dynamically*. Applications using D3P react at run-time to changing end user needs or execution conditions and according to constraints described in application policy statements.

1.3 Motivations: Data Exchange in Distributed Applications

Distributed applications and end users interact by sharing data, information, and even devices. In scientific endeavors, for instance, researchers remotely access resources like microscopes[8], 3D displays[128, 40], and even wish to operate sophisticated components like the Tokomac fusion facility. In industry, companies share parts designs[30] or other data critical to their operation. Examples include Schlumberger’s oil exploration processes

where reservoir simulation data produced in computer centers should be shared with 'on site' personnel conducting drilling[140], and where simulations should use well logs to refine current drilling procedures. example is the airline industry, as with Delta Air Lines desire to share flight and passenger information with third parties who distribute such data to select passengers for cellphone-based passenger notification[124]. Finally, in remote sensing and control, radar or camera data or telemetry/biometric information is captured in order to be forwarded to and used by interested parties, sometimes involving remote control loops as in telesurgery, targeting, and telepresence.

In applications like these (and our example), remote users are not interested in all of the data all of the time, and their interests can change rapidly. In fact, such *dynamic interest changes* often help make the implementation of such systems or applications feasible, or they are used to optimize implementations. Consequently, there are many models for such changes, including context sensitivity[48] in human-centered ubiquitous applications, spatial or temporal locality in pervasive and distributed systems[168, 9], and current focus or viewpoint in remote sensing, graphics, and visualization[94]. Finally, whether implicitly determined or explicitly captured by quality of service expressions[136, 137, 5], the occurrence of dynamic interest changes in applications and systems is accompanied by the wide range of effects they can have, starting with simple changes in data selectivity applied to ongoing information exchanges[94], continuing with the need to apply varying transformations to data[102], and also including real-time control reactions as in dynamic sensor repositioning or in telepresence[29] or teleimmersive applications[139].

The target systems and applications addressed in this dissertation capture, distribute, transform, and filter continuous data streams, in order to make appropriate data available where and when it is needed. The importance of addressing data management may in fact be critical to the advances sought by these applications. For such data streams, our research attempts to bridge the gap between the essential security support provided by authentication and encryption on the one hand and the low-level, often insufficient data management mechanisms offered by standard middleware and operating systems on the other hand. The specific problem we address is straightforward. End users typically organize

data to suit their various applications’ needs, often not with protection or security in mind.

In effect, *data protection may require orthogonal and complementary data organization, distribution and access patterns than the performance or ease of use considerations typically inherent in such applications.* Examples of data organizations for protection are (1) the need to give certain users access to some but not all of the data being transported or stored (e.g., removal of access to passenger information, but granting access to their food preferences), (2) the need to protect some of the data while sharing other information, as when cooperating researchers are happy to share ‘older’ or ‘low resolution’ results, but reserve ‘new’ or ‘high resolution’ results for distribution after publication, (3) the need to differentially protect certain data or certain transformations (i.e., services) performed on data, as when genome data needs to be protected due to patenting or intellectual property restrictions, and (4) the need to differentially secure certain data, as when sensor images may contain some highly secure data (e.g., persons’ faces, identified military objects) that must be ‘fuzzed out’ or ‘blacked out’ prior to distribution to others. In all such cases, security-based data organization is orthogonal to or sometimes even contrary to data organizations required for suitable performance in transport and access.

For any given data stream, a principal problem we address is how to protect and secure certain data in that stream, distinguished by data type (e.g., ‘passenger id’ field of the ‘passenger’ event) or even data content (e.g., data values and positions associated with face recognition). Furthermore, we are also concerned with the use of stream manipulations by specific end users; these manipulations represent differential access to the data stream. In summary, we are concerned with the provision of differential data protection for dynamic data applications, without unnecessarily disturbing original data organization and structure.

1.4 Threat Analysis

A protection mechanism implies the existence of threats against it. Several frameworks for modeling threats in detail exist [108, 141, 87]. At this point in our presentation, a threat can be usefully modeled as a combination of an asset, a vulnerability, and an attacker. Also important are the failure modes made possible by particular threats and how they are

addressed by the mechanism.

The basic asset protected by D3P is the structured data exchanged by distributed applications. However, such data has application-level semantics associated with it. Not simply structures, the data is a stream of images, a description of aircraft parts, or a collection of molecules. Asset confidentiality (limiting access to the data or knowledge of its content or semantics) and asset integrity (preserving the structure and content of the data) are orthogonal but important aspects to consider.

The vulnerability of this data is wide-ranging. Consider the case of a sensor providing a data stream. An attacker could gain unauthorized access to the data stream, or the ability to corrupt it, in several ways:

- acquiring physical control of or access to the sensor, the network, or the hardware at each application node;
- subverting the kernel- or user-level software of the operating system on any application node; or
- subverting the application itself.

Attackers in D3P are both malicious and not so. Malicious attackers seek access to protected data in contravention of a defined security policy, or to covertly convert data to a form where it does not conflict with any such policy. Non-malicious attackers are those who might inadvertently expose access to a data stream (by publicizing a stream subscription point or installing the wrong data conversion).

Potential failure modes for applications in domains addressed by our implementation of D3P are of two types: attacker access to data where such access is disallowed by policy, and access to functionality where such access is similarly disallowed. Specific failure conditions for functionality deployment can include application or node failures; incorporation of user-specified code into an application can result in explicit termination of the application, access to application data not protected by D3P mechanisms, exploitation of application identity on particular hosts, and system-wide failures such as denial-of-service attacks engineered by infinite loops.

Middleware-level implementations of D3P obviously cannot address vulnerabilities relating to physical security of hardware devices, processing units, or network elements. Remedies for such attacks are straightforward. D3P also does not address operating system level attacks where “rootkits” or other commonly known OS vulnerabilities result in superuser-level access to all user processes on a node. D3P protects against the use of the middleware against itself, regulating access to middleware interfaces and guarding against inadvertent data exposure. Also, by relying on a trusted code repository and verifying user code integrity, D3P guards against “malware”-style attacks. The philosophy of D3P, and therefore its approach to potential threats, is well approximated by Anderson in [13]. Speaking of cryptosystems, Anderson claims that designers focus more often on what could *possibly* go wrong rather than what is *likely* to, and further that things likely to go wrong are issues of implementation and management. D3P is not a cryptosystem, but the point is equally applicable.

Also, the most serious threat to computing systems frequently stems from principals that already have some legitimate level of access (“insiders” as referred to by Summers [152]). Rather than guarding against random operating system vulnerabilities exploited by anonymous hackers, or staking out another point in the data space of systems aiming at provably complete protection, D3P directly addresses the insider threat. Insiders in the application domains we consider are the scientists who wish to collaborate but also wish to preserve proprietary data, or the providers of sensor data who wish to implement tiered access for performance and cost control. They are the users of the applications, the designers of the data being exchanged, and D3P directly addresses a valuable subset of their data manipulation and protection needs.

It is, however, important to note that the D3P model is flexible enough to allow alternative implementations. Such implementations can make use of different technology to counter threats left unaddressed by the work described here. For example, current complementary work proceeds on integrating safe kernel-level extension [66] and kernel-level communication [106] mechanisms with the D3P approach to policy specification and component-based

application structure. This work will provide efficient data transport while isolating potentially dangerous code segments from the application being adapted. Another implementation of D3P concepts targets data transfers on serial buses inside the kernel [113]. The implementation we present in this dissertation is a first step in exploring how the concept of D3P can be used to protect applications against threats at several different levels.

1.5 Thesis Statement And Dissertation Organization

This dissertation makes the following claims:

- An infrastructure for providing data differentiation and protection is a useful system design asset in the high-performance and pervasive domains, making possible applications whose functionality and security requirements would otherwise make difficult successful application development and deployment.
- Such an infrastructure can be implemented with small, amortized performance impacts on large-data applications in those domains, and in some cases can improve performance over alternative solutions.

Our defense of these claims in this dissertation is structured as follows. In Chapter 2, we examine representative applications, one each from the high-performance, pervasive, and enterprise computing domains, and explain how D3P addresses their data security and performance requirements. Chapter 3 presents details of our reference implementation in a mature high-performance publish/subscribe middleware, and describes how that middleware is used by our sample applications. We also generalize the concepts that comprise D3P, with the aim of giving application developers a model that can be applied outside the areas explored in this dissertation. We discuss other pieces of the infrastructure necessary to support application design and development with D3P in Chapter 4. In Chapter 5, we present a first benefit of applying such a model: providing protected access to application configuration audit trails. We present experimental results supporting our claims for D3P performance implications in Chapter 6. Chapter 7 discusses related research not covered elsewhere in the dissertation, and we present closing commentary and outline directions for

future work in Chapter 8.

CHAPTER 2

APPLICATION DOMAINS

D3P is designed to address modern distributed applications, which are characterized by their long lives, dynamic constraints and needs, and component-based nature. In particular:

- Distributed systems and applications are becoming increasingly long-lived. Sensor systems continuously collect, stream, and analyze data. Global applications like grid services must stay online around the clock to meet the needs of international scientific or business processes. Therefore, changes in data protection, or more generally, any such application-level adaptations, cannot be accommodated with methods that require system downtime. For dynamic data protection, this means that the methods that implement them must use general solutions for in-place, online system and application evolution, such as dynamic code generation, runtime extension, and specialization [28, 131].
- User/device needs and environmental constraints are dynamic. As a result, applications are written to dynamically accommodate new user- or domain-specific functionality, implemented by user-supplied or second party codes. Dynamic data protection, therefore, must support runtime changes in application structure and in the data being transported and manipulated by applications.
- Applications are component-based, and not all components are statically known. Modern applications are neither deployed nor maintained in a monolithic manner. They make heavy use of dynamically loaded library codes. New application paradigms like peer-to-peer [99] allow for an arbitrary or unknown set of users. Reflective/introspective development frameworks [116, 93, 115] and open standards for control transfer [35] and data description and exchange [2] have made it difficult for application developers

to know *a priori* about all communication types or methods and the range of execution environments used by applications. For dynamic data protection, this implies that dynamic configuration changes in applications must be addressed as a first-class design goal.

The primary contribution of D3P is that it can deal with middleware systems along each of these three axes *differentially*. That is, where previous systems have only allowed their users or developers coarse-grain ways of dealing with changing user populations (to either allow unknown users or not), data definitions (preventing the use of low-display-capability hand-held devices in rich scientific visualization settings), or restricting dynamic functionality to a set of pre-loaded binary libraries, D3P systems can make differential decisions, according to application security policy. For example, users who are not allowed by application security designers to view particular fields in a data type can be prevented from seeing exactly those fields, rather than being restricted from seeing the type altogether. D3P provides a method of avoiding such all-or-nothing decisions.

The threat model faced by such applications is varied. *Active* threats, where actors either known or unknown to the application take specific, deliberate actions to gain unauthorized access to data, damage data integrity, or modify system configurations in order to degrade or halt application execution are possible. For instance, user-specified code that purports to perform a given task but in reality modifies data could be loaded as an application extension. *Passive* threats involve mostly loss of confidentiality of information in an application, rather than its integrity or damage to application execution itself. In applications that communicate via streaming data, as do those we consider here, this threat is most commonly realized through unchecked or unauthorized subscriptions to data streams. *Execution-based* threats such as denial or theft of application services are also possible; for example, users may be able to subscribe to data streams that have been processed by methods not paid for (e.g. face recognition in image streams) or supply an adaptation that stops application execution (a shared object file whose code contains an infinite loop).

D3P directly addresses threats in these categories. Active threats are countered by maintaining specific policy information regarding users and system extensions, and governing which users can install which extensions. Extensions are assumed to be verified by means outside the D3P system (formal verification or inspection), but once this verification has taken place and the extension code signed, D3P guarantees only that particular code can be loaded by any authorized user. Eavesdropping is not possible in a D3P system, as any access to a data stream must be checked against security policy. Subscription to a data stream requires authorization by a trusted policy module. There is also no way for threat actors to browse application information (such as the collection of data types available); users are given access to exactly the data types required for their execution needs and no others. Finally, the trusted policy manager used in a D3P-based system only allows registration of application adaptation code by known and authorized users, reducing the chance that malicious adaptations can be introduced. Theft of application services is prevented by supplying users with references to data streams containing “finished” data, where authorized actions (such as face recognition) have already been applied, and disallowing unchecked stream subscription or adaptation.

The D3P model relies on the integrity of the trusted policy manager. If this process is subverted or successfully impersonated, the data protection mechanism cannot provide the guarantees we have discussed. Cryptographic techniques (e.g. shared-secret identification) are used to confirm the identity of the policy manager, and stronger identification methods can be introduced into the D3P model without loss of generality or functionality. Proof against subversion is provided through operating system facilities, the presumption of code inspection/verification involved in any software module assumed trusted, and hard-coded auditing tools which produce records of the module’s actions.

The following sections examine the need for and utility of D3P in three representative applications: (1) a surveillance system that uses remote cameras to capture and inspect images; (2) a high performance application used by scientists to collaborate in real-time via meaningful remote data visualizations; and (3) an enterprise-scale system where multiple business partners require different views on the same data.

2.1 Pervasive Applications: Surveillance

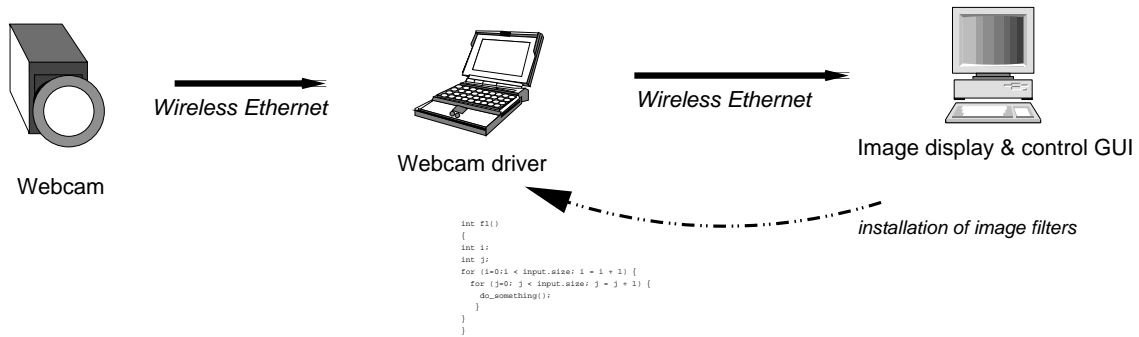
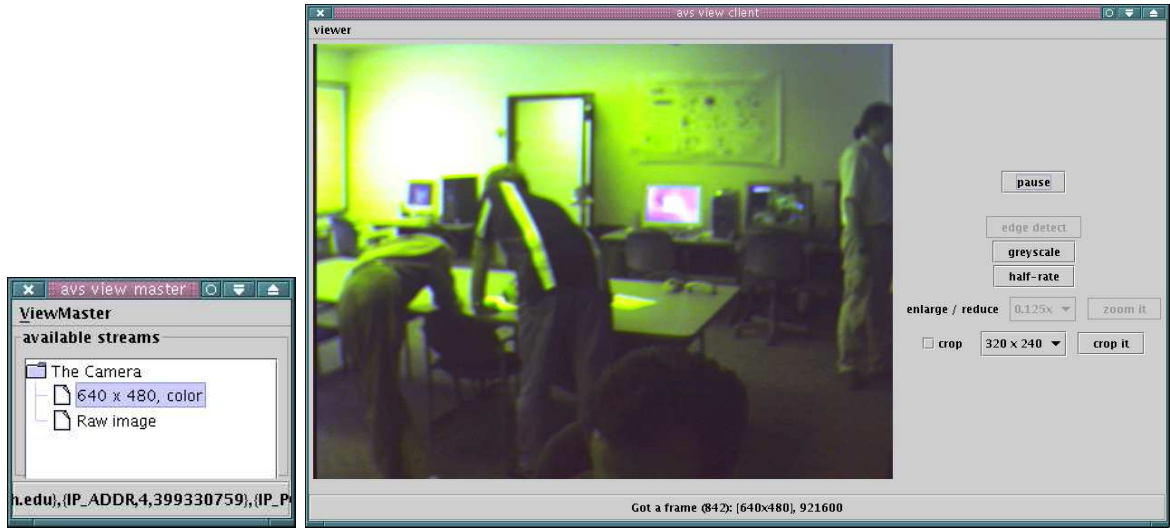


Figure 2: The AVS (Active Video Streams) application.

Consider the simple surveillance application depicted in Figure 2, which shows the component architecture of the Active Video Streams (AVS) utility developed by our group. AVS [171] emulates the basic functionality of remote sensor-based applications. AVS typifies such applications in that it streams data captured by a remote sensor, exemplified by a camera, and transmits them to interested consumers. To enable rich image analysis, uncompressed data is transferred using the PPM industry-standard image format, as 640x480 PPM image frames, approximately of size 960Kb each. In our simple demo application, these frames are consumed and displayed by a Java image viewer that emulates the control panel used in a surveillance application. In generalizations of applications like these, analysis functions are applied to incoming image data before it is displayed, automatically or as needed (e.g., initiated by surveillance personnel) [3].

A basic trade-off in remote sensor processing, especially across the wireless communication platforms for which AVS is intended, is the delay in data transmission from sensor to viewer vs. the quality of sensor data received and analyzed. AVS provides multiple sensor-resident data filters that implement suitable trade-offs. Specifically, AVS allows each consumer to customize the image stream by dynamically introducing a data reduction (filter) function into the data path.

Among these actions are *greyscale*, where the RGB color image is downsampled to a greymap; *resize*, where the image capture size is reduced by 50 or 75 percent in both



(a) Available video streams.

(b) A viewer displays a video stream and gives access to possible adaptations.

Figure 3: Command Center Interface.

dimensions; and *crop*, where a portion of the image is selected by the user and the remainder discarded. Each of these functions implements a different trade-off in the amount of data transferred across the wireless network link and therefore, the delay in data transfer vs. the utility of the data received by surveillance personnel. Whether customized or not, image frames are represented and transmitted as structured data packets as shown below.

```

#define AVSIMAGE1C      921600 /* 640 * 480 - color */

typedef struct {
    int tag;
    char ppm1;
    char ppm2;
    int size;
    int width;
    int height;
    int maxval;
    char buff[AVSIMAGE1C];
} Raw_data1C, *Raw_data1C_ptr;

```

Figure 4: C language structure representing a PPM image used by the AVS application.

AVS provides a rich environment for defining security policy, where usage semantics and access policies for users, devices, customizations and streams must all be coordinated. D3P addresses these needs as follows:

- The primary access control decision for AVS is whether a user can view images on

a particular stream. AVS has a type system that is used for selection of streams — users choose to view a 640x480 color image stream as opposed to a 320x240 greyscale stream, for example, and these streams are implemented using different data types. D3P permits policy to leverage the type system. That is, D3P supports access control decisions by restricting access to the type and thereby the image data.

- D3P enforces access control decisions by issuing capabilities for data and types, thereby removing from developers the need to make authentication and authorization decisions. Instead, with D3P, one simply presents the capabilities provided by the user to the middleware. If the capabilities allow the requested access, it is granted. This frees the AVS developer to concentrate on building the best image display and manipulation application possible, instead of worrying about access control.
- Concerns about the mechanics of installing user-directed adaptations are also relieved by D3P. Using the same capability model that governs access control, installing adaptations on image streams simply involves presenting the appropriate capabilities to the middleware. Integration with the type system is also implicit, as the same capability access model is used. An example is downsampling, where a user wishes to improve frame rate by reducing the amount of data transmitted. In AVS, the user can install an adaptation on a color image stream which produces a greyscale image stream. D3P reduces this operation to the presentation of the appropriate capabilities for the existing stream, the adaptation, and the data types involved. Installation of the handler, retrieval of the new type information, and construction of the new image stream are all performed automatically.

This analysis shows how the D3P approach to middleware can have direct and immediate benefits for application developers in the pervasive domain. Similar advantages can be enjoyed by high-performance computing applications, as we discuss in the following section.

2.2 High Performance Computing: Data-centric Collaboration

Consider the exchange of technical data between collaborating scientists or engineers. Applications in which such data exchanges take place are those built with the SmartPointer framework for real-time scientific collaboration developed by our group [174]. As depicted in Figure 5, scientists can share visual displays of output data generated by a high performance simulation, they can view data across heterogeneous network links and on different display devices, and they can select at runtime the subsets they wish to view and/or analyze of the large output data sets generated by the running simulation.

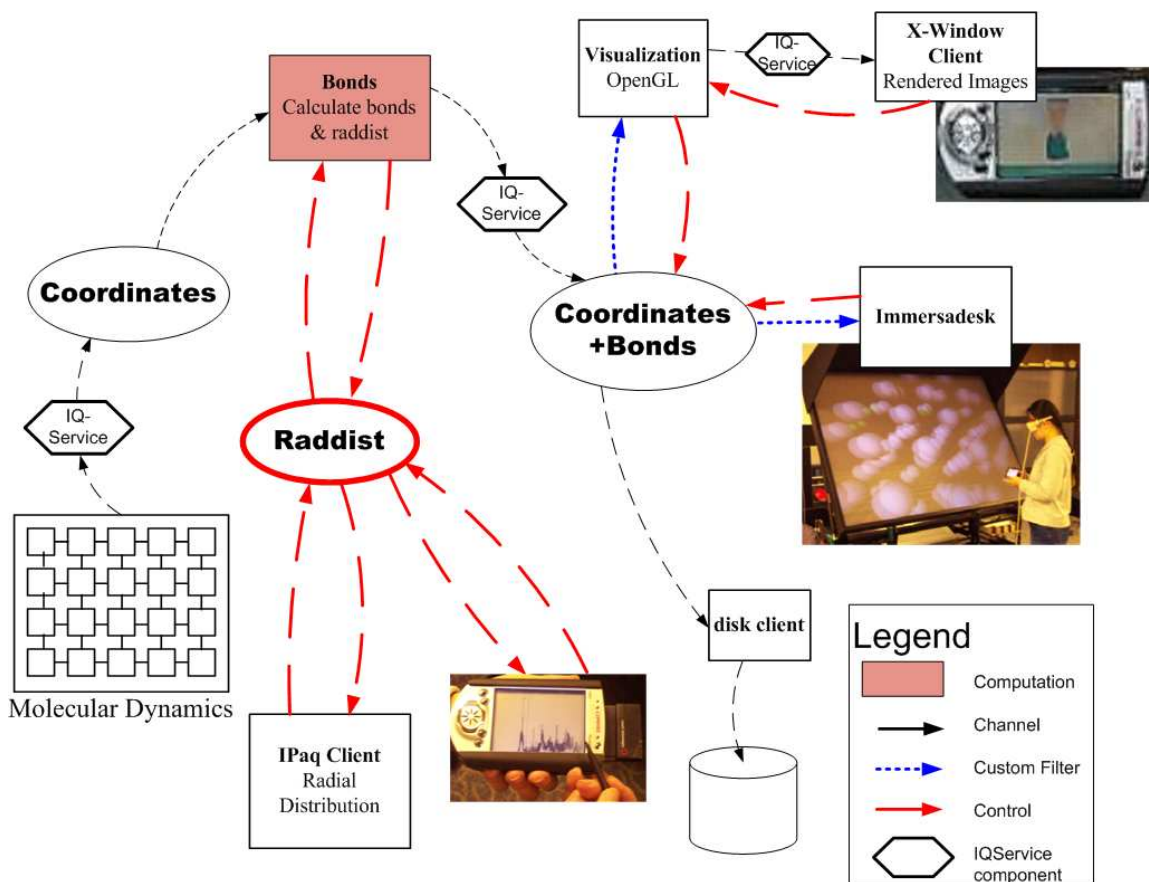


Figure 5: The SmartPointer application. Scientific users can share information about the molecular dynamics simulations on heterogeneous output devices ranging from wirelessly-connected handhelds to those with advanced 3-dimensional rendering capabilities.

Two concrete examples of desired data sharing are: (1) a scientist viewing only the copper atoms (and their behaviors) simulated by a running molecular dynamics simulation vs. (2) another scientist interested only in the chemical bond data computed from simulation output by running certain analyses across that output. In both cases, middleware represents simulation output as well as the products of output analyses as structured data types. The data stream produced by a simulation may be customized with dynamically generated code in the same manner as AVS image streams. Some of these customizations concern access restrictions. Such restrictions are particularly important in large-scale collaborations like DOE's ongoing Supernova Initiative, where a multi-disciplinary team of scientists is investigating the complex processes ongoing in a supernova explosion. Here, conflicts arise between the necessity to collaborate in order to make progress and the desire to protect "proprietary" data and/or methods. Controls over data sharing and flexible access policies are vital in these cases. Similar scenarios occur in industrial collaborations, where subcontractors with technical expertise in particular areas need carefully-specified and monitored access to valuable engineering data.

As with the AVS example above, D3P makes possible solutions that directly address these issues. In the specific case of the SmartPointer application:

- The data exchanged by the components of SmartPointer are represented as structured data types. These structures may contain both simulation output and analytical results. D3P uses the structure definitions provided by the users of the application to define fine-grained access policies. Users are not required to think about security restrictions, other than in terms of the structure of the data they are generating and analyzing (with which they are intimately familiar!).

It should be stressed here that the users of the application are *not* primarily software developers and do not necessarily have experience with the middleware used to implement SmartPointer. In effect, D3P leverages users' experience with the structure of the data they are generating and analyzing. D3P provides the means to directly leverage that experience when performing access control.

- SmartPointer and similar applications are used by diverse groups of researchers, separated by geography, administrative structures, and research domains. Control over data access and system customization is made much more difficult when users belong to such decentralized groups. Using D3P-capable middleware moves the expression of control over data access (i.e., the definition of security policy) out of the application and into more-easily-verified trusted policy modules. This is done in a decentralized fashion using D3P capabilities, the latter being abstract, revocable, and portable instruments.
- A key attribute of applications like SmartPointer is that end users are the ones who express the ways in which data streams should be customized. D3P directly supports user-driven ways to extend or adapt applications, independent of geographical and organizational considerations. The approach is to integrate the type definitions provided by researchers, their customization code, and security policy (likely defined by non-researchers) into a single system. The mechanics of system adaptation work identically for all users, and security decisions are encapsulated into capabilities. The efficiency of data transfer is preserved by the use of dynamic code generation.

Consider the threats to data protection involved in such applications. A passive actor could gain access to proprietary supernova simulation data simply through the application itself by simply browsing the available data streams. Active threat actors could seriously degrade the execution of vital simulations by introducing denial-of-service adaptations. In an unprotected SmartPointer application, all users would have access to all data types provided by the application and be able to use them to synthesize new streams. D3P explicitly guards against each of these types of threats, by requiring authorized access to data streams, data types, and adaptations.

2.3 Enterprise Applications

The past several years have been a period of considerable upheaval in the enterprise systems area. In order to reduce costs and improve service offerings, major information systems

integration efforts have been undertaken. These efforts often include attempts to tap into the vast amount of real-time and near-real-time data stored in legacy databases, and re-purpose or otherwise manipulate that data in support of new functional or competitive requirements.

Such an effort, described in [124], is the construction of a Operational Information Systems (OIS) mid-tier at Delta Air Lines. OIS (Figure 6) is designed to “mine” the legacy online transaction processing infrastructure, deriving new business data from existing processes with minimal disruption. Scalability of OIS is a primary concern, as the number of endpoints for such data is potentially in the hundreds of thousands.

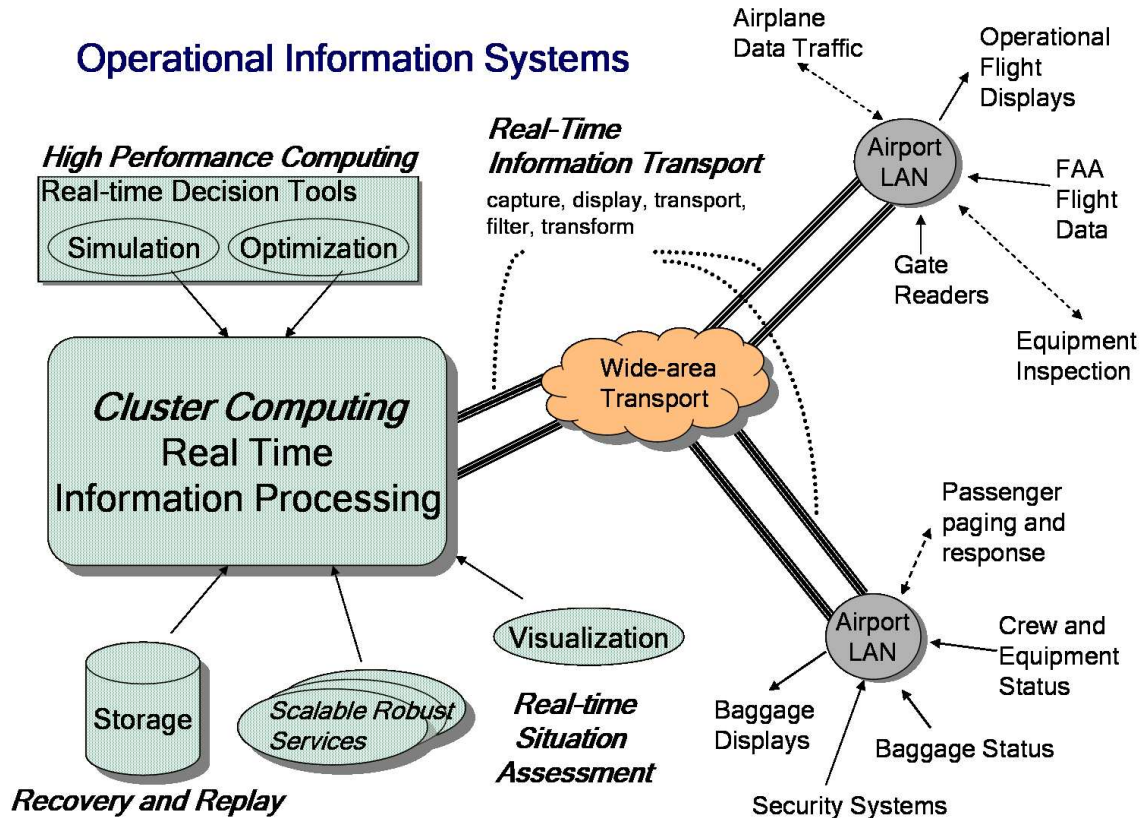


Figure 6: Architecture of the Operational Information Systems mid-tier.

The primary purpose of OIS is the dissemination of existing or newly synthesized data. However, even among data intended for the same class of endpoints (business partners such as caterers, for example), there are data sharing issues. Caterers should not have access

to entire passenger records; meal preferences are valid data to pass along but names and addresses are not. Fuel vendors at airports might need specially synthesized records in order to properly provision local fuel services. Law enforcement agencies may need on-demand access to customized data streams derived from OIS.

D3P provides unique and novel facilities to address these concerns:

- Many middlewares ship unstructured data in enterprise applications such as OIS. However, the rise in popularity of XML has increased the incidence of structured data. Where structured data and sufficient metadata exist (as in XML-based middleware or hybrid solutions [144] which switch between text and binary message structure and content), D3P can provide flexible data protection. Developers of the “data taps” that comprise OIS-like applications can create D3P policy independently of other data streams being generated from the same source.
- The set of interested parties for enterprise data is large. Business partners, consumers both inside and outside the enterprise, and external agencies such as regulatory bodies or law enforcement organizations may all have valid data inspection or usage needs. Creating policy for such diverse groups of users would be extremely problematic in a development environment that did not support decentralized policy definition and management. D3P-enabled middleware allows decentralized establishment of policy in rapidly-changing enterprise information system architectures. Separate policies acting on the same data stream can be simultaneously developed and deployed for both law enforcement agencies and caterers, without requiring data to be duplicated.
- The attractiveness of the OIS “data tap” philosophy is that multiple new applications can be developed on an *ad hoc* basis without disrupting existing data streams. Synthesis of new data streams required by new applications is accomplished by injecting application-specific functionality. D3P directly supports this paradigm. D3P further provides unrelated application development teams the ability to define their own metadata and policy information. This keeps development impact small and increases the flexibility and responsiveness of new applications.

- OIS is fundamentally a middleware solution to the problem of developing and deploying new enterprise applications. D3P provides middleware interfaces for modular development of such applications and as such is correctly targeted for OIS. Furthermore, without the ability to incorporate user-defined policy provided by D3P, many OIS applications would not be realizable at reasonable execution costs.

In this chapter, we discussed the application domain which D3P addresses and the types of protection threats D3P does and does not provide leverage against. We presented example applications from each of the pervasive, high-performance, and enterprise computing domains. In each case, D3P makes possible unique and novel solutions to the problem of managing data-centric application development.

CHAPTER 3

AN IMPLEMENTATION OF D3P IN PUBLISH/SUBSCRIBE MIDDLEWARE

In order to explore application development with D3P, a concrete implementation of D3P concepts has been created with a mature, high-performance middleware library. We note, however, that the concepts of D3P are suitable for a wide variety of middleware. The key requirements are the availability of a type system (through either metadata or reflection) and the ability to dynamically associate functionality at a link endpoint. We choose to implement D3P in *publish/subscribe* middleware for two reasons: (1) the classes of applications we target make heavy use of publish/subscribe concepts, and (2) we have at hand a mature publish/subscribe middleware package called ECho.

In this section, we present interesting details of a concrete implementation of D3P principles. Among these are the necessity to control access to middleware objects, the management of application functional and security policy, and implementing differential data protection through the application of user- and policy-specified functionality to data streams. We ground our discussion by first describing the fundamentals of the middleware library we use as a starting point.

3.1 ECho Fundamentals

ECho [52] is data delivery middleware for the high performance and pervasive domains, targeting interactive scientific collaboration, remote instruments and visualization, and similar large-data applications. Superficially, the semantics and organization of structures in ECho are similar to the Event Channels described by the CORBA Event Services specification[76], implementing an anonymous group communication mechanism. Data senders in anonymous group communication are unaware of the number or identity of data receivers. Instead, data is delivered to receivers according to the rules of the communication mechanism. In this

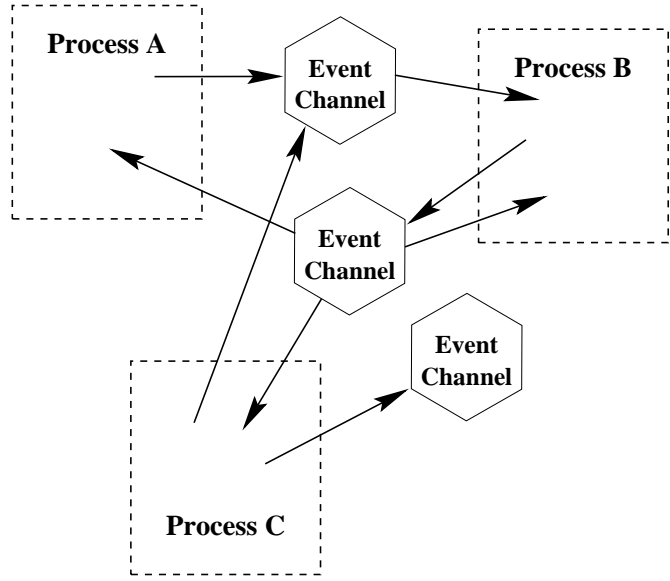


Figure 7: Processes using Event Channels for Communication.

case, event channels provide the mechanism for matching senders and receivers. Data messages (or *events*) are sent via *sources* into *channels* which may have zero or more *subscribers* (or *sinks*). The locations of the sinks, which may be on the same machine or process as the sender, or anywhere else in the network, are immaterial to the sender.

A program or system may create or use multiple event channels, and each subscriber receives only the messages sent to the channel to which it is subscribed. The network traffic for multiple channels is multiplexed over shared communications links, and channels themselves impose relatively low overhead. Instead of doing explicit *read()* operations, sink subscribers specify an upcall to be run whenever a message arrives. In this sense, event delivery is asynchronous and passive for the application.

Event channels are distributed entities, with bookkeeping data in each process where they are referenced. Channels are *created* once by some process, and *opened* anywhere else they are used. The process that creates an event channel is distinguished in that it is the contact point for other processes wishing to use the channel. The channel ID, which must be used to open the channel, contains contact information for the creating process as well as information identifying the specific channel. However, event distribution is not centralized and there are no distinguished processes during event propagation. Event messages are

always sent directly from an event source to all subscribers.

ECho channels can be either typed or untyped. The type system for typed channels is managed by PBIO (Portable Binary I/O) [23]. PBIO allows high-level description of data to be efficiently represented and transmitted in binary form. PBIO transparently handles binary translation issues such as differing machine word sizes or word endianness, provides facilities for compile-time or run-time type definition, and performs type reflection and conversion between compatible types.

3.2 Overview Of Basic ECho/D3P Usage

An application E using ECho to transmit or receive data follows a general form. E creates a ECho context object, uses that object to obtain a handle for an event channel, and then establishes access to the channel as either a source or a sink. An application D using the D3P version of ECho follows a similar pattern, but with some additional steps involved. We will now describe this process in a general fashion, in order to provide context for more detailed discussion to follow later in this chapter.

The most significant change is communication with a trusted security and policy management service called *Overwatch*. An Overwatch daemon executes on each host in a D3P application. When D performs the ECho library call to create a context object, it now must supply some form of authentication token that the local Overwatch can understand. The library call in this case, into a version of ECho that has been D3P-enabled, relays this token to the local Overwatch instance. The local instance verifies the token (in an implementation- and policy-dependent manner) and responds with a verification message. This message contains enough information for ECho/D3P to construct the context object and return it to the caller.

This *protected* context object is then used, either directly or indirectly, in all subsequent ECho operations performed by D. If D is an event subscriber, the context object is used to acquire a channel handle and then subscribe as a sink. In this case, the D3P middleware extracts the protection information from the context object and supplies it to Overwatch

as proof of the calling application’s identity. Overwatch verifies that this protection information is valid and consults an internal policy brokering engine to determine which, if any, specific channel handle should be provided to D. In making this decision, Overwatch acts pursuant to application policy and functional information provided to it by application and/or system administrators. ECho/D3P receives a correct handle from Overwatch and constructs a suitable channel object for the application. That channel object can then be used to subscribe for data events.

This overview has been abstract in terms of the objects actually manipulated by such an application D. In the following section, we discuss concrete realizations for these objects and the requirements imposed by the design goals of D3P.

3.3 Implementing Protected Objects

In this section we discuss the implementation of protected object references for D3P. After presenting those details, we consider the differing protection guarantees provided by the D3P model and its reference implementation.

3.3.1 Protected References to ECho Objects

Ordinary ECho objects are C-language structures, produced by unprotected library routines. In order to implement the D3P protection actions described above, it is necessary to devise some method of controlling access to the creation and manipulation of those objects. The initial design for this task involved a “wrapper” library which completely encapsulated ECho and performed access checks on a per-function call basis. Once access rights were inspected, execution was delegated to the proper ECho library call. This approach was abandoned due to the amount of code that would have to change in applications already using ECho.

Adopting a design goal of minimizing source code changes to the ECho API (and therefore to applications), we choose to modify ECho objects themselves. To do this, we adopt the *capability* as a guiding abstraction. Canonically defined by Dennis and Van Horn[47], a capability can be abstracted as a unforgeable combination of a reference to an object with rights applicable to that object. In capability-based systems, all references to objects are

made through capabilities, and no access is allowed or even possible outside the capability mechanism.

Proof against forgery is provided in different ways, depending upon the level at which capabilities are used. Early systems such as the Chicago Magic Number Machine [177] and the Cambridge CAP [119] computer used capabilities to refer to memory segments, and prevented forgery by requiring that capabilities and user data never reside in the same segment. The Hydra [109] operating system kernel broadened the scope of capabilities to refer to all objects in the system (including both physical resources such as disks and I/O devices and logical resources such as files, processes, and procedures). Hydra’s capabilities were software-based, and capability integrity was enforced by the kernel (requiring frequent and costly domain switching). IBM’s System/38 [110] and Intel’s i432 [34] architectures moved these operations to the microcode and silicon levels, respectively.

This research aims to use capabilities in user-level software, to protect user-level objects. These objects are simply bit patterns in user memory, and to completely protect them would require either a machine architecture designed from the ground up to support capabilities [34, 173] or complicated memory protection arrangements. D3P’s aims are different, in that high performance and application-specific adaptability are primary goals; our intentions in the protection world are to explore enough of this design space to justify further research into more complete protection solutions. Consequently, the protection against capability forgery in this implementation D3P middleware is straightforward: language-level opacity is used to prevent direct programmatic manipulation of ECho objects, and protection against forgery takes two forms. The first is to ensure that a capability is *authentic*; that is, its reference and rights information can be regarded as genuine or approved by policy. The second is to ensure that an authentic capability cannot be modified. We implement the first form through signatures that use shared-key cryptography. Specifically, a ciphertext generated using an encryption key belonging to the service that signs capabilities is installed at the same time as ECho. This ciphertext is included in each successful capability request and then embedded in the capability. ECho then verifies the authenticity of the capability by comparing the capability ciphertext against the known ciphertext. To

guard against rights manipulation, a digest of the ECho object is created when a capability creation or modification request is granted or when a revocation occurs. This hash is verified when the capability is used in order to ensure that rights have not been unexpectedly changed.

This implementation of D3P is subject to vulnerabilities involving user-space code and data access. Ongoing research provides ways in which these vulnerabilities of the D3P implementation can be removed, and there are several research efforts [38, 32] which can improve the security of D3P implemented in user-level software. Also, as noted previously, other research [66] by our group aims to provide more complete protection mechanisms while still supporting the application performance and adaptability goals of D3P.

3.3.2 ECho Object Modifications

The “bootstrap” ECho object is the `EControlContext` (ECC); all other ECho operations require its cooperation. For D3P, the API for creating an ECC is modified to send a list of attributes to Overwatch. Overwatch responds to a successful request with a list of attributes containing its signature (to indicate success and to identify which Overwatch service responded) and a list of object rights for the new ECC (whether it can be used to create channels, open already-created channels, or both). This attribute list is then installed into a newly created object, converting the newly created ECC object into a capability. A flag is also set in the object to indicate that it is a capability; if an application attempts to use it in a non-D3P ECho library call, the middleware will spot the flag and refuse to continue the operation. Protected ECC objects can be used to create other protected ECho objects (which are themselves modified to act as capabilities).

Each ECho object creation API verifies that the ECC capability provided is valid and allows the creation of the object in question. For example, the event channel object in ECho is `EChannel`. The `EChannel_create()` call examines the provided ECC, verifying both the object signature and that the ECC contains a “create-channel” right.

The PBIO library also provides a set of objects used to represent data types and collections of those types. Capabilities for PBIO types are also obtained in cooperation with

the Overwatch process, in a manner similar to ECho objects. Data protection policy may only grant an application knowledge of a certain set of data types. P BIO defines a local dictionary of types known as an `IOContext`. Event submission in unprotected ECho requires an `EControlContext` and a specification of the type of the event. The analogous call in ECho/D3P is to provide capabilities for both the `EControlContext` and the `IOContext` containing the type in question.

3.3.2.1 Implementation Of Capabilities

The set of information that makes an ECho object into a capability is stored in an `attr-list`. `attr-lists` are supported by a separate library in the ECho software stack and provide serializable representations for arbitrary collections of data. Each item in the `attr-list` is called an `attr`. Typical `attrs` required for D3P capabilities include:

- The Overwatch ciphertext used for validating a capability;
- A rights mask for the object. For example, an `EControlContext` can possibly indicate rights for creating a channel or opening a channel; and
- A digest of the successfully created object's content (the default hashing algorithm is MD5 [135]).

The definition of each ECho object used as a capability was changed to include an `attr-list`. We use attribute lists in part to promote compatibility between future different flavors of D3P. It is conceivable that different implementations of the Overwatch service might provide multiple signature representations, each of which might be stored in a single attribute list. Although our current work does not address delegation of capabilities outside a single address space, we expect this flexibility to pay dividends in more distributed D3P environments.

3.4 Implementing Per-Client Customization

In this section, we describe ECho's facilities for performing per-client customization of event delivery. We then describe how the D3P implementation makes use of these facilities.

3.4.1 Customizing Event Delivery In ECho

ECho directly supports client customization of an event channel through the association of functionality with a link endpoint. Customization of an ECho channel in such a manner results in the creation of a *derived* channel (Figure 8). Derived event channels execute the function supplied during their derivation at the event source. The result of the function controls whether the event is sent to any subscribers of the derived channel. A critical issue in the implementation for ECho's derived event channels is the nature of the function and its specification. Since the function is specified by the client but must be evaluated at the (possibly remote) source, a simple function pointer is obviously insufficient. There are several possible approaches to this problem, including:

- severely restricting the function specification language, perhaps to a set of relational, equality and logical operators;
- using a generic, language such as C, but relying on pre-generated shared object files; or
- using interpreted code, like Tcl/Tk [125] or Java [107].

Having a relatively restricted filter language is the approach chosen in CORBA Notification Services [77] and in Siena [26]. While this facilitates efficient interpretation, the restricted language may not be able to express the full range of conditions useful to an application, thus limiting its applicability.

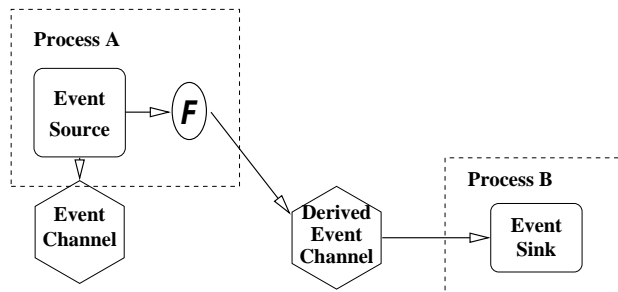


Figure 8: Derived event channels are created by applying a function to the output of an existing event channel.

Once agreed that a more general programming language is a better alternative, a remaining question is its translation in heterogeneous environments. One might consider supplying these functions in the form of a shared object file that could be dynamically linked into the application or an event source. Using shared objects allows these functions to be more general, but requires the client to supply them in a native object file for each possible destination. This is relatively easy in a homogeneous system, but becomes increasingly difficult as heterogeneity is introduced.

In order to avoid problems with heterogeneity one might supply such functions in an interpreted language such as a Tcl or Java. This would allow general functions and alleviate the difficulties with heterogeneity, but it would impact efficiency. Because of our focus on high performance computing and since most handlers we have found are quite simple, we have chosen a different approach that maintains high efficiency at some price in flexibility. We express functions in ECL, a subset of C, and resort to dynamic code generation to create efficient native versions of such functions on the target host. ECL may be extended as future needs warrant, but currently it is a subset of C, supporting the C operators, `for` loops, `if` statements and `return` statements. For experimentation, however, and in order to consider potential useful extensions to ECL, we have also enabled the use of general C-based shared-object modules.

ECL's dynamic code generation capabilities are based on Icode, an internal interface developed at MIT as part of the 'C project [129]. Icode is itself based on Vcode [54], also developed at MIT by Dawson Engler. Vcode supports dynamic code generation for MIPS, Alpha and Sparc processors. It has been extended to support MIPS n32 and 64-bit ABIs, Sparc 64-bit ABI, and x86 processors¹. Vcode offers a virtual RISC instruction set for dynamic code generation. The Icode layer adds register allocation and assignment. ECL consists primarily of a lexer, parser, semanticizer and code generator.

Figure 3.4.1 shows an example of a handler extracted from the SmartPointer application2.2. The handler computes (and passes) the average size of molecules in a given region.

¹Integer x86 support was developed at MIT. Vcode was extended to support the x86 floating point instruction set (only when used with Icode).

```

{
  int i, j;
  double sum = 0.0;
  for (i = 0; i < 37; i = i + 1) {
    for (j = 0; j < 253; j = j + 1) {
      sum = sum + input.molecule_size[j][i];
    }
  }
  output.average_size = sum / (37 * 253);
}

```

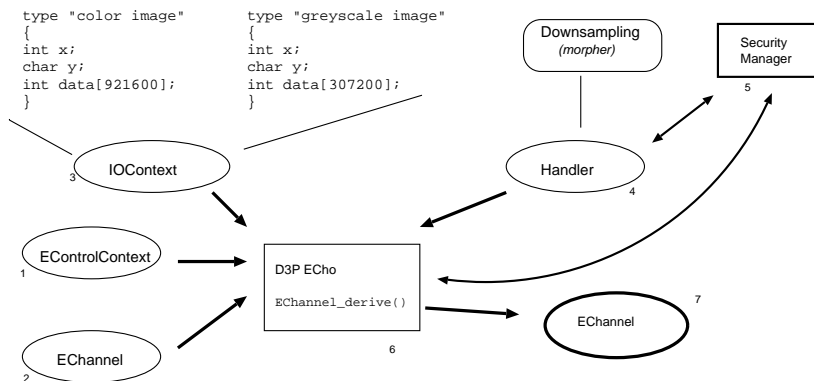


Figure 9: Installing a handler in AVS requires the use of several D3P objects. An *EControlContext* (1) is needed in order to create any other protected ECho objects. Capabilities for the original, unmodified *EChannel* (2), the PBIO types involved (the type used in the original channel and the one to be generated by the newly installed handler, in an PBIO *IOContext*) (3), and for the handler itself (4) are required for the handler installation. Communication with Overwatch (5) is necessary to retrieve at least the handler capability. The result of the operation (6) is a new protected *EChannel* object (7) which can then be used by AVS.

3.4.2 Using ECho Derived Channels

In order to create a protected derived channel, capabilities for the *EControlContext*, the original *EChannel*, any types involved, and the handler must be provided. The capability for the handler is also obtained from Overwatch; the user specifies a list of attributes of the desired handler (to execute locally or at a remote location, whether a DLL version is necessary, and other characteristics) and Overwatch acts as a broker to provide the correct handler code. Also, the *EChannel* capability must contain a “can-derive” right, indicating that the holder of the capability has the right to install a handler on it. The result of this operation is another *EChannel* capability which refers to the newly derived channel.

3.4.3 Providing Efficient Data Protection Through Type Manipulation

D3P provides data protection through type manipulation. In the strongly-typed, publish/subscribe application space we are considering, not having access to the data type provided by a particular service is functionally equivalent to not having access to the service. Without access to the type of data being transmitted, there is no way to reliably unmarshal data for use by the application. Providing differential access to services for applications becomes a question of providing differential access to strongly-typed data.

The D3P mechanism directly addresses this by requiring capability-based access to all middleware services. Forcing applications to acquire capabilities provides a level of indirection which we use as a hook to enforce policy. If application policies dictate that Alice should not have access to high-resolution sensor data, her attempt to acquire a capability from Overwatch for a channel carrying that data will fail. If policy statements forbid applications with certain attributes to install data transformation handlers on a particular channel, any attempt to acquire a capability from Overwatch for that handler and channel will fail.

It would be possible to customize streaming data services for individual users or applications by providing a completely separate stream each time one is needed. Consider Alice's situation described above; she is prohibited from accessing high-resolution sensor data (call this type H), but assume for a moment that she is allowed to see lower-resolution data (type L). A naive design for this situation establishes two separate data streams, with appropriate access controls for each. This wastes network capacity, as data is duplicated for the low-resolution stream.

D3P provides true differential access to the data in question. In order to prevent Alice from seeing prohibited data, Overwatch instructs the middleware to install a handler designed to customize the data stream in exactly the right way. This customization converts the data type from H (naming and containing high-definition sensor data) to a new type L that potentially has no naming or structural similarity to the first. In fact, Alice is not even aware of the existence or structure of H, since the middleware in conjunction with Overwatch substitutes a reference to a channel providing L data without her knowledge.

She cannot use the middleware facilities to convert L data to or from H.

Key to the efficiency of differential data protection is that the handler executes at the source of the data, minimizing data duplication and thus preserving network bandwidth. Any other application users in the same user class as Alice (that is, with access only to type L) will be given a capability to the channel originally established for Alice, further reducing bandwidth and processing wastage.

3.4.4 *a priori* Channel Customization

The attractiveness of application design in a D3P environment stems from the ability to establish, at design-time, a set of customizations that may be commonly needed. For instance, in the AVS application, the different image-processing algorithms are supplied as handlers which can be invoked by users or by Overwatch as necessary. This removes from application developers the need to take explicit action to ensure compliance with security policy. Contrast this with the need for UNIX programs that are installed with administrator privileges to explicitly give up those privileges as soon as possible to prevent security breaches. D3P middleware never transmits data to a user that should not see it, because protection actions are enforced by handlers that execute at data sources.

Application design then becomes more manageable in several ways. D3P promotes component-based application design, by encouraging the movement of functional elements into handlers. Applications become easier to reason about as a consequence. Data protection actions are removed from the application and encapsulated in the trusted Overwatch module. Finally, application policy is no longer implicit and static in the design of applications, but is made explicit and dynamic by the interaction between policy designers, Overwatch, and D3P middleware.

3.5 Overwatch, Application Metadata, And Policy Management

D3P is a data protection and manipulation mechanism, designed to be driven by application policy. The design intention was to remain *policy-agnostic*; as long as there is an Overwatch policy manager that can sign capabilities for the middleware, according to the rules laid out

by application policy, the specific implementation or behavior of Overwatch isn't important. The D3P model is intentionally vague on this point, to allow for the development of different kinds of Overwatch modules encapsulating different approaches to data protection policy management. In this section, we describe the operation of our prototype instance of Overwatch.

3.5.1 Overwatch Basics

Overwatch is a distributed service that runs on each host participating in a D3P-enabled application. The executable itself is a Simple Object Access Protocol (SOAP [35]) service constructed using the gSOAP [159] toolkit. Applications running on a particular host attempt first to contact the local instance of Overwatch; if a local instance does not exist or is incapacitated, a remote instance may be contacted. If no Overwatch instances are available, D3P services are not provided and applications are so informed through their ECho object creation calls.

The primary responsibilities of Overwatch are to (1) authenticate users based on credentials they present, and (2) determine which requests to grant based on such authentication and policy statements registered by applications and component developers.

Authentication is performed by comparing a username/password pair in an initial request to the information available from `/etc/passwd` on the system where the receiving Overwatch instance runs (typically this is the local instance of Overwatch and thus the local password file). Alternative authentication mechanisms include custom challenge/response schemes or PKI-based approaches. In a PKI-based system, the application provides an X.509v3-compliant identity certificate[86], whose signer is checked against an internal list of trusted certification authorities.

Authorization (deciding which requests to grant capabilities for) is a more complicated operation. It is implementation-defined, depending on the intersection of available resources, operations, and policy statements about principals. In our reference implementation, Overwatch uses initial metadata about local applications and extensions in conjunction with information about users. This metadata is maintained dynamically by administrators, and

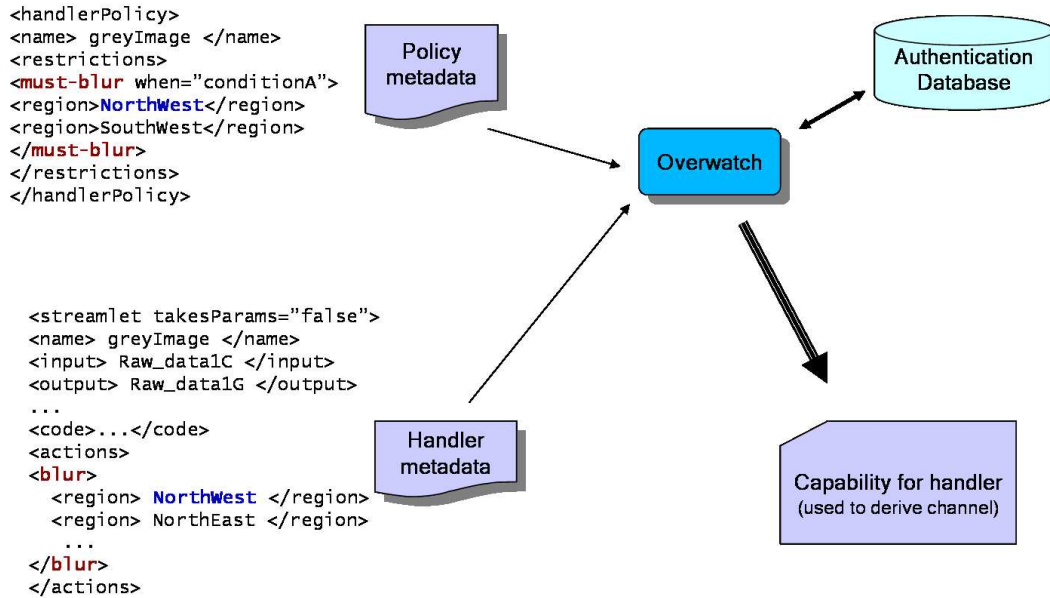


Figure 10: Use of metadata in the AVS application.

is used to generate handler capabilities for use by D3P applications. In the following section, we describe the structure of this metadata.

3.5.2 Metadata Used by Overwatch

Overwatch is initialized with two sets of data. The first is a collection of profile information corresponding to applications and their users. These profiles contain policy statements for users relative to data types and transformations available for each application. For instance, a profile might indicate that Alice is restricted from seeing an array of high-resolution sensor data contained in a particular application data type. The second is a description of transformations available for each application and actions that can be taken.

Consider as an example the set of actions available in the AVS application. One potential action is to blur or otherwise obscure a region of the image. This is implemented as a mathematical operation on the image pixels, which can be as complicated as a customized blurring algorithm or as simple as setting each pixel in the affected area to 0. Defining “affected area” is also crucial here; AVS works with rectangular images and so adopts a compass-oriented quadrant system. Figure10 illustrates this process in more detail.

In this example, a policy applicable to all AVS users is defined in which the “north-west” and “south-west” regions of the image must be obscured when condition *A* holds (the `must-blur` element in the XML policy document). In a fully-realized D3P application, condition *A* would be something obtained from the current environment or execution context of the application. Examples of such conditions include “the camera is pointing between 20 and 90 degrees”, “the time of day is between 0600 and 1200”, or “the installation security status is elevated”. The D3P infrastructure assumes the ability to acquire such information dynamically and with small delay. This is an area where proactive directory services assist greatly; Section 4.2 describes such a service more fully.

It should be noted that, although this example deals with a security-oriented scenario, this same approach can be used for instances more related to application functionality. For example, instead of policy metadata referring to a security action (`must-blur`), consider a policy designed to make AVS more usable by weak-display endpoints. Instead of a `must-blur` predicate, `must-downsample` might specify that any stream subscription should have its images downsampled from color to greyscale (thereby reducing the data transmitted by two-thirds assuming image representation as in AVS). When users request capabilities for image streams and this policy is in force, a downsampling filter is installed without user intervention or even user knowledge. From the display-challenged user’s point of view, the only streams available are those that are determined by policy administrators (presumably in consultation with developers) to best accommodate their display and connectivity. In this manner, Overwatch “differentially” supports vital application functionality.

Handler metadata is the other type of metadata required by Overwatch. This metadata describes the operation of a handler, which data types it requires, and which Overwatch-brokered actions it supports. For each handler used by an application, a metadata fragment like that depicted in Figure 10 is supplied to Overwatch by administrators. The code for the handler can either be provided in-line as ECL, or a URL can be supplied for a remote location where ECL or a binary object can be obtained. For brokering purposes, the `actions` fragment is most interesting. Each sub-element of `actions` has a name that is used to match requested actions in policy metadata. As in the example, specifying `must-blur` in the policy

statement will trigger a search for handlers claiming to implement the `blur` action in their metadata statement. If no such handlers are found among those registered with Overwatch, any requests for capabilities depending on them will fail² Each `blur` action defines a set of parameters, in this case the regions of the image that this particular handler is able to obscure.

When Overwatch receives the request for a capability, the principal making the request is first authenticated as described above. Then, the set of application policy statements is searched for items applicable either to all users or the principal making the request. If any handlers are required to be pre-installed, Overwatch includes in its response a directive for the middleware to install the handler and create a new ECho channel, returning the new channel to the application. Overwatch's signature is then applied to the resulting ECho object and the object's handle is returned to the caller.

Developers creating extensions for applications such as AVS must provide definitions and XML schema for the entire `actions` subelement of the metadata. In turn, policy writers can select from among the available actions of the available handlers and construct application policy metadata accordingly.

3.6 Toward a Model for D3P Application Design

In this section, we present a more abstract system model of D3P concepts. Our discussion to this point has been couched in terms of the publish/subscribe paradigm. By abstracting from those implementation details, we show that applications can be designed using D3P principles without being dependent on any particular messaging strategy. Figure 11 summarizes the model design.

3.6.1 Model Objects

In the D3P model, computations are carried out at *hosts*. These computations are encapsulated in *tasks*, an arbitrary number of which may execute at any host. Tasks communicate

²Other implementations of Overwatch might choose to forward such requests to other Overwatch instances in the hopes of locating a suitable registered handler.

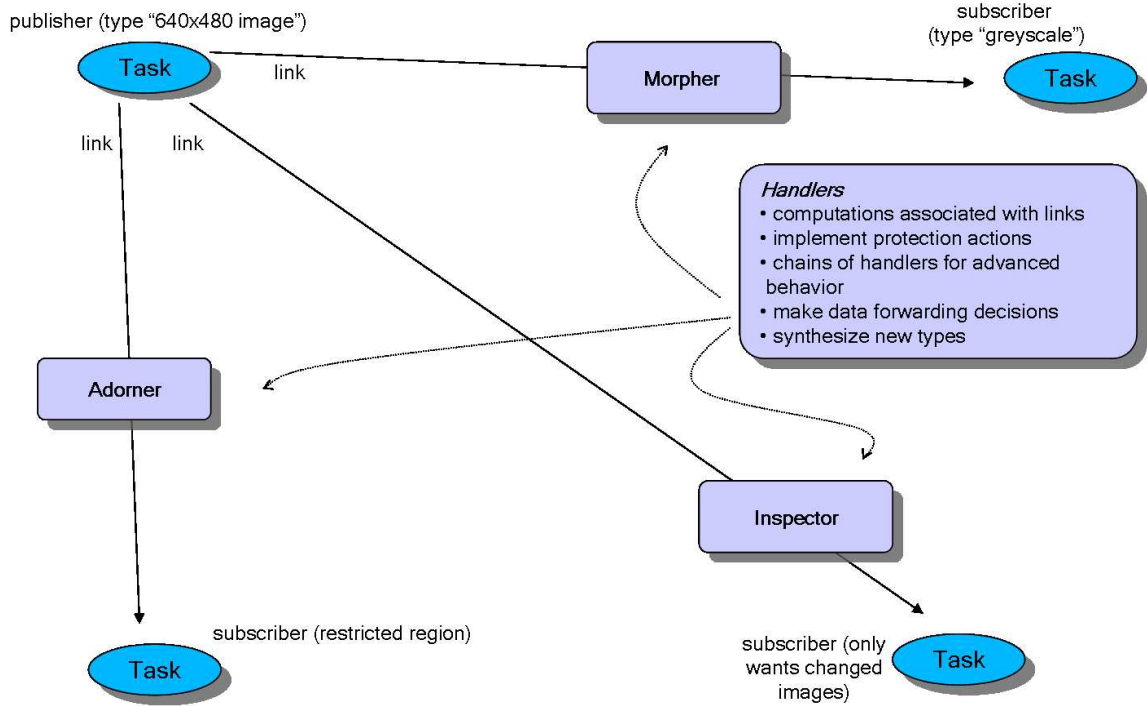


Figure 11: Illustration of the D3P system model.

with each other over *links*, which are unidirectional and asynchronous. Information transmitted over each link has a *type*, and each link carries information of exactly one type. It is possible for a task to discover the structure of any unknown type by contacting a type-server. Any task may introduce a new type at any time.

Other computations can be associated with links; such computations are called *handlers*. A handler is logically associated with a task at either endpoint of a link (i.e., at a host) and is executed when a task sends or receives data. Handlers accept input and may or may not produce output. They have access to the type information of the link with which they are associated, and can perform stateful, content-aware actions based on the information in the link. Handlers always accept an input type (the type of their link), and always produce a boolean value. They may also produce an output type (potentially any type in the type universe). Handlers also may change their input data. We say that handlers are themselves typed; this type is actually a 2-tuple (I,O), where I is the input type to the handler and O is the output type (or `nil` if there is no output type).

Handlers can differ in the way they treat their input data and produce output data. These differences are reflected in the three classes of handlers: *inspectors*, *adorners*, and *morphers*. *Inspectors* accept a specific data type as input and produce only a boolean result. These handlers may perform some type of reflection on the type or computation using the data itself to determine the result value. For example, the handler could be an expression of criteria that the input type or data must satisfy. Or, an input type that contains an array of floats might be vetted by a handler that computes the arithmetic mean of the values in the array and compares it against a given amount. *Adorners* produce an output type — the same as the input type. However, Adorners are so named because they may make changes in their input data. Revisiting the Inspector example above, an Adorner could not only compute the average of a subset of input data, but also store it in the output data for transmission along the link (assuming, of course, that the Adorner returns a `true` value). *Morphers* add the ability to produce an output type that is different from the input type. They provide the ability to perform type specialization or narrowing based on input type information, input data, or data computed by the handler itself.

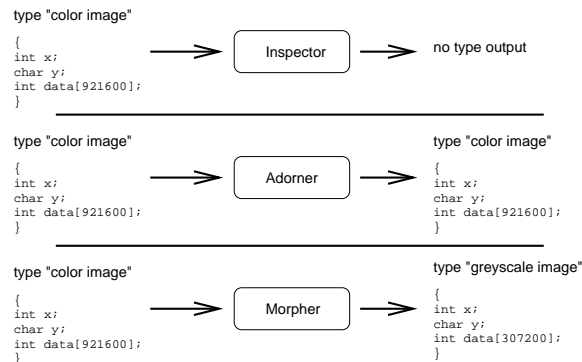


Figure 12: Inspectors, Adorners, and Morphers take different actions on input and output types.

If multiple handlers are associated with a link, they execute serially in the order of their time of installation. When installing a second handler, the input type of the new handler must be the output type of the existing handler. A set of handlers at a link endpoint is called a *handler graph*, and is a directed acyclic graph with a distinguished single *head* handler and a distinguished single *tail* handler. Handler graphs, as with single handlers,

have a type which is expressed as a tuple. A handler graph type tuple consists of the input type of the head handler and the output type (or `nil`) of the tail handler. The boolean result value of the graph is the value provided by the tail handler.

An important property of links with installed handlers is that the data is only forwarded to the destination endpoint of the link if the handler returns `true`. If the return value is `false`, the data is discarded. For handler graphs, their default organization is a set of logical expressions across the graph. By default, each of these is a logical AND across the results of each path in the graph from the head to the tail. In this mode, for each path through the handler graph, the boolean return value from each handler must be `true` in order for the remaining handlers in the path to execute. If each handler in a path returns `true`, and the tail handler executes and returns `true`, data is forwarded across the link. For purposes of determining whether or not data is forwarded, the graph is treated as a “black box”. The complete boolean expression represented by the “black box” of the handler graph can be modified by inserting different boolean and grouping operators after the graph is constructed.

Complex functionality can be encoded and manipulated using graphs of connected handlers. In such graphs, data is projected from a handler into one or more downstream handlers. Data can traverse the graph along multiple paths according to application requirements. Any symmetric data manipulation task (such as encryption/decryption) can be modeled in D3P as the installation of matched pairs of handler graphs at appropriate link endpoints. More advanced strategies can include “slicing” data so that particular pieces can execute at tasks with specialized abilities (for instance, special hardware support for rendering images being shipped as part of an event). Several research efforts are directed at working with such graphs. One such is the Active Streams programming model [25], which provides guidelines for defining and manipulating diverse handler (*streamlet* in that context) structures. Active Streams supports simple point-to-point operations as well as complicated split-and-join behavior (which can be useful in heterogeneous network environments).

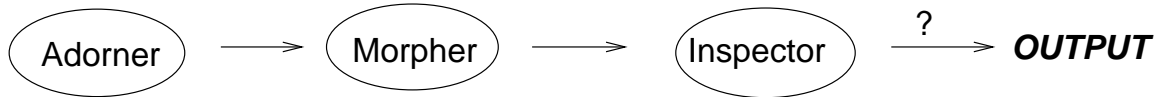


Figure 13: Handlers can be combined in *graphs* to achieve a series of effects.

3.6.2 Capabilities

All D3P objects are manipulated through a single mechanism: capabilities. Generally, a capability combines an unforgeable reference to an object with an expression of the set of permitted operations for that object. D3P capabilities are created upon request by a trusted model object called Overwatch. Overwatch encapsulates policy decisions and manipulation, handles the publication and updates of capability revocation lists, and removes the necessity of distributing trusted components throughout the model. Capabilities contain references both to the object they name and to the owner of the capability; both references are necessary to execute operations in the protection model. All model operations require a capability with appropriate rights. Construction of a link requires a special capability granted by ; installing a handler on a link requires a capability referencing the link that has “install-handler” rights, as well as a capability for the handler that permits installation.

The “black box” view of a handler graph provided by D3P allows graphs to be manipulated and reasoned about in the same manner as single handlers. A handler (or graph) and the link to which it is attached can also be referred to as a compound object, using a single capability. This property of D3P objects prevents an explosion of capability references. It also makes possible the advance definition and packaging of interesting functionality.

As mentioned earlier, links are strongly typed. The type system is also manipulated through the use of capabilities. In order to install a handler that refers to a particular type, a capability for that type must be presented. If a Morpher is being used to perform type conversion, capabilities for both the input and output types are necessary.

The ability to “prepackage” links and handlers provides D3P with an important part of its usefulness. Links can be predefined with handlers that implicitly convert types to those authorized for particular users. Data access can be provided on a differential, per-endpoint

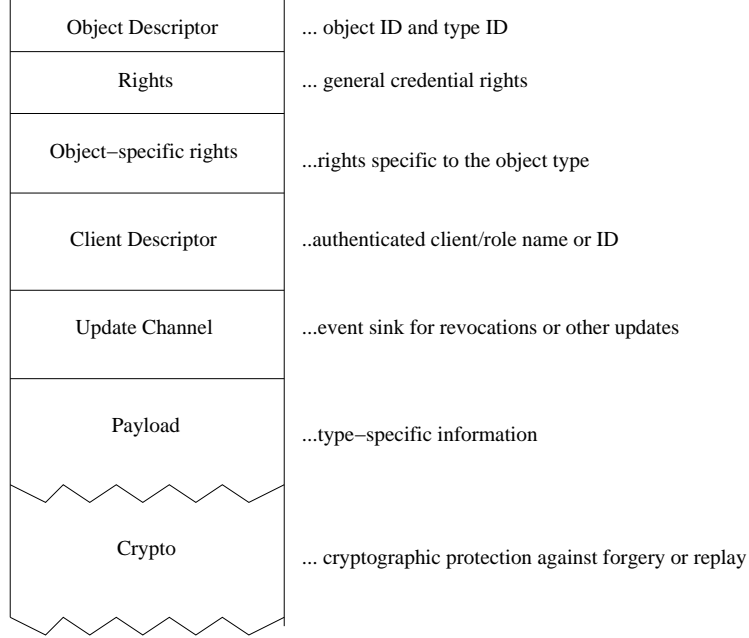


Figure 14: Capabilities in D3P can be viewed as having record structure.

basis, where each endpoint for a link is constructed with a customized Morpher based on policy information obtained from Overwatch.

3.6.3 Type Definitions

Although a full type-theoretic treatment of D3P has not been completed, we present several definitions here for clarity. There is a *universe* of types available in the model which we denote \mathcal{U}_{data} . These are the types which applications use to represent and manipulate their data. It is actually more appropriate to say that there are two such universes, where the second is $\mathcal{U}_{primitive}$ and corresponds to the set of primitive or atomic types used for composition. Although it might seem that $\mathcal{U}_{primitive} \subset \mathcal{U}_{data}$, the D3P model exchanges structured composite data and so the two sets of types are disjoint. \mathcal{U}_{data} is defined by the set S of type-servers available to the application. Any type-server in S may contribute a type to \mathcal{U}_{data} , and any application may use any type $T \in \mathcal{U}_{data}$.

Handlers operate on types in \mathcal{U}_{data} . As specified above, handlers themselves have types, which are drawn from the universe $\mathcal{U}_{handler}$. The members of $\mathcal{U}_{handler}$ are defined as $T_1 \rightarrow T_2$, $T_1 \in \mathcal{U}_{data}$, $T_2 \in \mathcal{U}_{data} \cup \mathcal{U}_{primitive}$. D3P defines three types of handlers in $\mathcal{U}_{handler}$. For a

handler H ,

- H is an Inspector $iff H : T_1 \rightarrow \text{boolean}$;
- H is an Adorner $iff H : T_1 \rightarrow T_1$; and
- H is a Morpher $iff H : T_1 \rightarrow T_2, T_2 \in \mathcal{U}_{data}$.

3.6.4 Degrees of Freedom

One measure of the usability of middleware is the number and type of restrictions placed on application designers and developers. We note that several degrees of freedom are preserved by the D3P model. First, D3P requires only a type system to support inspection of data or differentiate between multiple accesses to a single type. D3P is independent of knowledge of roles, principals, or other high-level security modeling concepts. Second, the only constraints on how handlers are expressed are the presence of a type system and the requirement to produce a boolean output. This allows a wide range of possible expressions for computations. Handlers can run the gamut from rule-checking engines such as are used for network firewalls to architecture-specific, dynamically-loaded binary code objects. Third, although handlers are logically associated with link endpoints (i.e., with sending or receiving tasks), D3P does not prevent implementations from decoupling handler execution and application-level endpoints. This property can be very useful in cases where specialized hardware is available at remote hosts or when network conditions require alternate data routing strategies. Finally, D3P allows arbitrarily complex structures of handlers to be defined and modified according to application requirements. These structures can take the form of simple chains, where output from each handler is routed to the next in a serial manner. They can also be complex graphs representing application workflow, hardware, or network characteristics.

3.7 Model vs. Implementation

The abstractions provided by the D3P model are designed to allow implementations whose implementation goals may be different from each other. The reference implementation of

D3P aims to provide a basic testbed and proofs-of-concept. As such, its design makes compromises with respect to the types of threats against which it protects applications.

Capabilities in the model are conceptually much closer to memory-guaranteed objects such as found in Hydra. They are portable, unforgeable and non-reproducible. This allows the model to make guarantees that the reference implementation cannot, and raises questions in the reference implementation that the model does not currently address. For instance, unchecked capability *delegation* is certainly possible in the implementation, since capabilities are implemented as user-space objects. The model restricts delegation to those capabilities with appropriate rights set. Implementation objects are simply memory regions that can be copied into other regions. The implementation does guard against moving capability information out of process, by including process and thread IDs in the capability attribute list and checking them at subscription time. A D3P application could make arbitrary copies of its own capabilities and use them interchangeably, possibly resulting in denial-of-service attacks (if sink subscriptions are doubled) or data stream corruption (if each event in an image stream were published twice, for example).

Capability *revocation* in the model also provides stricter guarantees than in the implementation. Revocations in the model happen directly between Overwatch and the capabilities involved; rights are changed directly and atomically, without any implied communication or action on behalf of an owning task, and the changes detected on the next use of the capability. Since D3P/ECho capabilities are C-language objects, they cannot themselves receive revocation notices. Since Overwatch is a separate user-space process, it cannot directly change the rights in a capability affected by revocation. A revocation event sink and capability revocation queue are implemented per process. Performing their processing may result in time lapses between the policy change that results in revocation and a change in application behavior resulting from the attempt to use a no-longer-valid capability. Such time lapses might possibly disclose unauthorized data, allow unauthorized or incorrect channel derivations, or otherwise compromise a D3P system.

Code isolation in the D3P model is not addressed completely. It is assumed that Overwatch will supply only handlers that are trusted to install and execute correctly. This

allows tasks in the D3P model to disregard the dangers of incorporating “foreign” code into their execution loops, guaranteeing freedom from one half of the *mutual suspicion* problem. The converse problem, regarding the safety of handler code or data from modification by the task, is not considered. In the implementation, handlers run in the user space of the application, and their data and code spaces may be inspected or modified. The converse problem also exists, as handler code has free access to the data space of the application. This gives rise to problems with data confidentiality and code safety. The implementation does guarantee that a particular handler will be verified as genuine before channel derivation occurs. Future research [67] is aimed at providing improved code isolation and sanitization guarantees in an efficient implementation.

CHAPTER 4

INFRASTRUCTURE COMPONENTS FOR DATA PROTECTION

There are several infrastructure components required in order for D3P to provide its services. In this chapter, we describe these components and how they contribute to the overall D3P picture.

4.1 Data Type Services

The modern distributed applications targeted by D3P exchange structured data that digitally describes mechanical parts or equipment under design, graphical objects being displayed, or scientific data representing atmospheric volumes and chemical concentrations. The infrastructure used by D3P provides efficient wire formats for that structured data through the use of PBIO [53]. It also separates metadata specification from data structure definition via the XMIT toolkit [170]. For D3P, this is important because XML-based type specifications are heavily used in Overwatch policy statements (in order to promote ease and flexibility of type definition). These packages are described in the following sections.

4.1.1 PBIO

PBIO (Portable Binary Input/Output) is a binary-format message definition and manipulation library. By using PBIO, distributed applications can use binary data in its native forms, by providing support that enables participating components to share metadata about data structure and layout. PBIO's *native data representation* (NDR) is a “receiver makes it right” approach, where the sender transmits the data in its own native data format, and it is up to the receiver to perform any necessary conversion. Any translation on the receiver's side uses custom routines created through dynamic code generation (DCG). By thus eliminating compile- or design-time rigidly defined common wire formats, components' data

exchanges can be updated whenever and wherever needed. In addition, PBIO avoids the up and down translations required by approaches like XDR. Furthermore, when sender and receiver use the same native data representation, such as in exchanges between homogeneous architectures, this approach allows received data to be used directly from the message buffer, thereby eliminating high copy overheads [138, 167]. When sender's and receiver's formats differ, NDR's DCG-based conversions have efficiency similar to that of systems that rely on *a priori* agreements to make use of compile- or link-time stub generation. However, because NDR's conversion routines are dynamically generated at data-exchange initialization, our approach offers considerably greater flexibility. The metadata required to implement this approach jointly with the runtime flexibility afforded by DCG allow D3P to offer XML or object-system levels of plug-and-play communication without compromising performance. Stated more explicitly, explicit metadata for type definition (defining the layout, and in certain cases sizes, of items within a data structure) allows very precise and targeted marshaling actions to and from wire formats. While this level of metadata is "hard-coded" and unresponsive to application changes under other middleware architectures, D3P provides methods to separate the explicit definition of type metadata from efficient implementation of the formatting decisions based on that metadata. These methods are encapsulated in the XMIT toolkit described next.

4.1.2 XMIT

An important property of mechanisms for structured data exchange is the degree to which they permit the independent evolution of data definitions, thereby enabling applications to communicate via enhanced data structures. This is difficult when using communication mechanisms like RPC or CORBA's remote object invocations that perform data definition in a programmatic fashion. Embedding metadata into communication or application code may result in good performance, but substantial costs arise when applications evolve, as changes in metadata require consequent modification and recompilation of the codes using such metadata. In addition, this approach limits the utility of metadata; while most systems

can be usefully abstracted by the structure of the data they exchange, the level of expressiveness of commonly used programming languages such as C is a severe handicap. Finally, embedding metadata also ‘hides’ it from exactly the non-programmer end users to whom it is typically most useful: the engineers designing parts, the scientists studying atmospheric phenomena, and others, who share data in their distributed collaborative workspaces.

Our research has been predicated on the belief that open metadata systems will become increasingly important and useful, especially for non-programmers using distributed computations that share substantial amounts of data. This belief is validated in part by the increasing popularity of metadata standards like XML [2]. However, the success of open metadata systems requires that their use does not unreasonably degrade the performance of applications that use them. Consequently, our work seeks to reconcile openness and performance.

4.1.2.1 Efficient Binary Transmission of Structured Data

Our approach addresses interoperability at levels ‘below’ those of RPC or CORBA, but with functionality exceeding that of common data exchange formats like XDR. Specifically, we are concerned with the efficient movement of the data structures that are defined at the ‘system’ level of distributed applications and middleware, typically using implementation languages like C. Such structured data usually resides in main memory, and when moved across heterogeneous machines, issues including byte-order, field alignment, and atomic type representation must be addressed.

Furthermore, at this level, data transmission in binary format is critical, due to the high communication bandwidths or low transmission latencies required, or because of the undue processing loads that would be imposed on systems if they were forced to transform information from end user readable formats, like text, to binary formats, for instance. Sample applications requiring binary data transmission include high performance codes moving scientific or engineering data and wide-area transfers of operational data, where scalability to many information clients and sources implies the need to reduce per-client or per-source processing and transmission requirements. They also include server-based

applications in which single servers must provide information to large numbers of clients.

4.1.2.2 Efficient and ‘Open’ Specification of Data Structure

In CORBA, data structures are defined using IDL specifications. In RPC, procedure parameters are characterized by their types specified within ‘interface module’ descriptions. Openness in metadata definition implies that data structure specifications are not linked to certain transmission mechanisms, such as RPC, or specific protocols, such as those used in the transmission of manufacturing, parts, or design information in the automobile industry, or as those used by specific data storage facilities (e.g., database query languages). Instead, openness requires that data structure may be specified independently of data transmission and use, with translations of such structure to the efficient lower-level representations used for data transmission or manipulation ‘hidden’ from end users.

We have developed a novel approach to open, high-performance systems. Our approach decomposes the transfer process for structured data, which results in efficient, binary (not text-based) low-level encodings of data, while also maintaining open user-readable and -comprehensible data structure definitions. We described our implementation of this approach, the XML metadata Integration Toolkit (XMIT), a tool which provides flexible metadata definition using XML, while also supporting high-performance, binary data transmission. We demonstrated that XMIT provides performance comparable to that of binary data transmission, by using runtime methods of establishing structured data exchange. Small ‘startup’ overheads are incurred only during ‘connection establishment’, that is, each time an XMIT-based exchange is initiated and/or the structure of the data exchanged is modified.

4.1.2.3 Open Metadata Specification In A D3P Infrastructure

D3P leverages the decoupling of data definition provided by XMIT in its specification of handler metadata. As presented in Chapter 3, metadata statements for handlers contain a specification of the data types used for input and output. This specification may consist of a string naming the type, in which case an internal lookup table of predefined type definitions is used. For additional flexibility, XMIT-style type definitions can be used. In

such cases, the XMIT toolkit is used to dynamically construct a type definition for use by a D3P application. The reference D3P implementation uses the P BIO library to define data structure, but other implementations of the Overwatch module may wish to use other wire-format definitions. Using XMIT, D3P applications can operate with Overwatch modules using any wire-format definition scheme.

4.2 Directory Services

D3P applies to a wide variety of distributed applications, particularly including loosely coupled applications crossing physical and organizational boundaries. Advances in communication technologies and the proliferation of computing devices have made this possible; two such types of infrastructures are pervasive computing environments [55, 74] and computational grids [160, 75, 58]. An important component of D3P, therefore, is an infrastructure layer that allows distributed resources and services to be pooled and managed as though they were locally available. A key element of such a layer 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) [57] for Globus-based environments, and the Intentional Naming System (INS) [7] for applications developed in the Oxygen [114] 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 [117], LDAP [176], and X.500 [132]). Recent work has addressed the issues of expressiveness of their object descriptions and query languages (through attribute-value hierarchies [7], for example) and considered their scalability (through domain-based partitioning or hierarchical organization [7, 42, 43, 150]). 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. [42] 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.

We have previously described how an *exclusively* inactive interface to directory services can hinder server scalability and indirectly affect the behavior of potential applications [149]. In [24], we propose extensions of 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. 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. In this section, we will describe the Proactive Directory Service and how it contributes to the D3P infrastructure.

4.2.1 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 [161] for zone change notification, the Ninja Secure Directory Service (SDS) [43] for service announcement, and the "persistent search" extension to LDAP proposed by Smith et al. [148].

Our extension of directory service 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

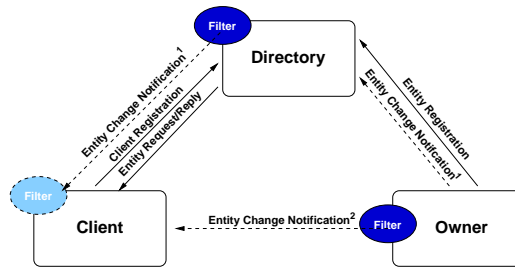


Figure 15: Owners register entities with the directory. Clients poll the directory for specific entities or types of entities. Clients can register to receive notifications of entity changes (from the directory (1) or the entity’s owner (2)). Clients customize notification channels through filters placed at the notification sources.

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 [27, 151] or in a general interpreted language such as Tcl/Tk [125] or Java [107]. 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.

4.2.2 Proactive Directory Service in a D3P Infrastructure

PDS is used by the Overwatch module to stay aware of policy changes. This is primarily useful in determining which conditions in policy statements might be applicable at any given time. Policy metadata includes a PDS server and namespace reference, which Overwatch uses to subscribe to an update channel. When context or environment information changes, Overwatch is notified by PDS and updates its internal condition table appropriately. This enables Overwatch to respond dynamically to application state changes.

A key issue in context changes, and one well-studied in capability systems, is that of capability *revocation*. What should be done when the conditions in force for the creation of a capability at time T no longer obtain at time $T + 2$ minutes? Recall that the definition of a capability is a reference to an object combined with rights to that object. Changes in conditions imply changes in the rights of the capability, to the extreme of rendering the capability entirely invalid. Obviously, new requests for capabilities can be handled in light of the changed execution conditions, but previously issued ones must be modified. The current implementation of D3P handles this by maintaining a revocation ECho channel

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

to which all D3P applications implicitly subscribe. When revocations occur, an event is pushed to this channel. The `EControlContext` object used by the application receives this event and determines which ECho capability must be modified. On the next use of the revoked ECho capability (attempt to subscribe to a channel as source or sink, attempt to derive a channel from an existing one, submission of data as a source or receipt of data as a sink), the capability's now-updated rights are checked and the appropriate action taken. In the most common case, where data is received by an application acting as a sink, the application event handler is not called and the application is un-subscribed from the ECho channel.

In this manner, an application context change monitored through PDS results in the dynamic adjustment of application access to data and/or functionality.

4.2.3 Broker Services

When starting, distributed applications must be able to discover the state of their peer network. In particular, applications sharing communication channels need to find which channels are already established and can be used and which remain to be created. We employ a *group server* to coordinate these activities. Other distributed systems have similar components - brokers in CORBA [1] or the JINI [116] architecture, for example. Applications contact the group server to find out which communication channel they should be using. If no such channel exists, the application is free to create a channel and register it with the group server.

4.3 Summary

The integration of the infrastructure components described in this chapter into the D3P programming model provides novel benefits to applications.

Extension of the D3P data protection model into the PBIO data definition library provides users with a high-performance binary data-formatting package whose metadata system is access-controlled. This control is exercised by the Overwatch policy module, using policy information derived from the same statements as are used to describe data stream and handler policy.

D3P's use of the XMIT metadata toolkit allows flexible use of XML Schema data types to be used in policy statements. Important to note here is that XMIT type specifications can be used to define new types, with policy information associated directly and immediately instead of being separated or coded into application logic.

D3P leverages the proactive abilities of the Proactive Directory service in order to receive dynamic notice of application policy updates. As application policies or execution conditions change, Overwatch modules on affected hosts can become aware of those changes and implement needed revocation or stream reassignment actions. Moreover, Overwatch becomes aware of these changes as an event consumer, without the need for constant polling. This also allows applications to define specifically what kinds of context updates they wish to implement.

Each of these infrastructure pieces is implicitly part of the trusted computing base of D3P applications. Further research into D3P concepts will explore how to reduce the size of this trusted base by integrating the D3P access control model more fully across the infrastructure.

CHAPTER 5

SUPPORTING AUDIT AND CONFIGURATION CONTROL USING D3P

D3P is designed to provide data protection and manipulation functionality. Its design stems from the idea that protection actions are better taken earlier rather than later. However, the mechanism used to implement D3P also allows the construction of systems which report on all adaptations and extensions, as opposed to ensuring that only the correct ones occur. While this approach is less secure, it is also easier to achieve. More importantly, the information flow created from reporting such actions can be a valuable resource.

The issues addressed in this dissertation are rooted in recent research trends toward large-scale, component-based distributed systems that are dynamically configurable or extensible in response to changing execution environments or end-user needs. Regardless of whether these configuration changes happen automatically through predefined adaptation or self-management methods or in response to explicit user interaction, they can jeopardize the integrity of application components. Moreover, they can cause unexpected effects in system performance or even lead to disputes about middleware or application providers' responsibilities for failures experienced by end users. This chapter introduces Reverb, a set of middleware abstractions and mechanisms derived from core D3P concepts. Reverb can be used to (1) audit configuration actions, (2) impose controls on permissible actions, and (3) control which principals are permitted to carry out configurations. We have integrated Reverb into the same middleware used in the high performance domain as we have used for D3P. The intent of this integration is to not only demonstrate its viability and utility, but also to show that Reverb-based configuration control has little effect on the performance of the distributed applications or middleware that use it.

5.1 Reverb Background

Industry efforts like IBM's autonomic computing [91] or HP's adaptive enterprise [83] initiatives are concrete demonstrations of the increasing importance of dynamically managing, configuring, and adapting distributed applications. New functionality being developed in these contexts and in ongoing basic research, and being integrated into distributed applications or systems includes self-repair in response to failures, resource awareness to better match application behavior to available platform or network resources, and similar methods. Supporting technologies that are making it ever easier to dynamically configure distributed systems include new middleware and system techniques like code injection, dynamic code generation, and runtime system extension.

The Reverb abstractions and their middleware realization described in this chapter address several problems shared by all adaptive systems: (1) how to audit or track dynamic configuration actions, (2) how to impose controls on permissible actions, and (3) how to control which principals are permitted to initiate and carry out configuration actions. With this functionality, Reverb addresses important needs of future self-* systems. Runtime code injection, for instance, can lead to problems when applications written with complex middleware infrastructures like Websphere or JBoss fail: should support personnel be responsible for fixing issues due to dynamically injected code, for instance? And, which company's support personnel should be responsible? Similarly, while it is already well-established that runtime system maintenance or upgrade should be done only by certain, responsible parties, how can we extend suitable controls on less radical runtime updates, e.g., those implied by shipping a new class structure to an application server in a J2EE infrastructure? Finally, for complex system infrastructures or applications, surely, it should not be the case that everyone can ship such classes to whoever needs to use them? Reverb addresses basic questions like those raised above, by coupling runtime *auditing* of configuration changes with *fine-grain controls* on what is being audited and on the set of configuration changes certain principals are permitted to make. Auditing and control are integrated into the middleware or the system mechanisms that are used to carry out configuration changes, where Reverb views all such changes as operations applied to the entities being configured.

Fine-grain constraints may be specified and enforced as to which particular changes are audited – *differential auditing* – and then, as to which principals are permitted to carry out which configuration operations on which entities, termed *differential change control*. Further, Reverb makes such operator/entity associations entirely dynamic, so that distributed applications can be differentially audited and/or controlled only when desired or needed by end users. This permits developers to focus Reverb on those application components or behaviors currently of most interest to them. Finally, by implementing differential audit checks and controls with dynamically generated binary codes, Reverb performance effects are felt only when Reverb is currently being used, thereby accommodating the high performance or more generally, highly resource-constrained applications and systems addressed by our research.

The Reverb implementation also comprises tools useful for developers and end users. Remote code repository, directory, and storage mechanisms may be easily associated with differential auditing or control runtime functionality. An XML-binary conversion tool may be used to convert XML-based audit and control policy specifications to efficient binary codes installed in applications or added to code repositories. Directory support is written to permit principals to change as and when necessary [24], and audit codes may be dynamically bound to additional functionality that produces audit logs for postmortem analysis of desired vs. observed configuration changes.

The Reverb approach to configuration forensics addresses explicitly executed configuration actions like runtime code injection, kernel extension, etc. It may detect changes caused by failures, implicitly, by comparing otherwise collected log information against Reverb forensic logs. However, it cannot prevent such changes, nor can it prevent changes caused by attacks. In fact, attackers could compromise Reverb's operations itself, thus still requiring systems to use intrusion detection or similar techniques to notice or discover attacks and/or compromised system components. A kernel-level implementation of Reverb now underway to track runtime system extensions is less vulnerable than the middleware realization described here, but there are many system-level attacks that can compromise its operation.

The applications addressed by Reverb are the complex distributed codes now being widely deployed, in Internet-based systems [93], in pervasive environments, or in wide area scientific collaboration. By tracking and then limiting and controlling the ways in which principals can change such applications, we hope to make it easier for developers, support personnel, or end users to differentiate appropriate from inappropriate changes, expected from unexpected change events, and reasonable from unreasonable change actions. Another outcome is the ability to establish a clear history of change events and to trace them back to the principals that made them, thereby making it easier for developers or support personnel to exploit developer skills. An interesting extension of Reverb is the provision of additional functionality that would detect or perhaps, even react to interesting event correlations, such as correlating actions resulting from specific user interactions with actions taken by self-changing applications.

This chapter describes a realization of Reverb in high performance publish/subscribe middleware used by data-intensive applications. The goals are to demonstrate the viability and utility of Reverb and to show that its presence and use do not impact the performance experienced by applications. Experimental results, presented in Chapter 6, show that when not being used, Reverb functionality has marginal effects on the fast paths of data transport and manipulation of high performance applications. This is in direct contrast to the costs experienced by persistence support, for example, in middleware like JavaSpaces [60, 181]. When being used, Reverb auditing can be quite inexpensive, especially when auditing is coupled with simple in-place analyses [80] to detect specific conditions or behaviors. As stated earlier, such analyses are dynamically created as binary codes generated from XML-based policy specifications. Audit logging overheads are proportional to the extent of in-place audit analysis and the amounts of audit data produced by logs.

Our current research uses Reverb to monitor configuration changes in a high-performance, large-data application and produce an audit trail that can be used for on-line or off-line analysis. Monitored changes include the creation of communication links, subscription for data, and the introduction of user-code into the data transmission path. We show that this monitoring can be accomplished in a flexible and scalable manner, with performance costs

amortized across application data exchanges and low run-time overheads. In this chapter, Section 5.2 motivates and describes the Reverb approach to handling configuration changes for distributed adaptive systems. Section 5.3 illustrates the effectiveness of Reverb in addressing configuration changes in the context of our experience with a sample adaptive application.

5.2 The Reverb Subsystem

In this section, we present an overview of Reverb and how it provides facilities to address system configuration changes.

5.2.1 Reverb Mechanisms

Reverb distributes information relating to configuration changes to interested (and optionally only to authorized) parties. Such configuration changes can include creation of a channel, subscription of a particular client to a channel as source or sink, and/or the introduction of user- or system-supplied code into an application. While use of other configuration mechanisms (such as `syslog` [4]) requires an explicit call from the program, Reverb produces information implicitly as a consequence of any configuration change.

This information is distributed as configuration events over a dedicated event channel (the RChannel). Reverb explores two novel extensions to this concept. First, Reverb provides the ability to control access to this channel, basing access control decisions on application-specific and system-wide policies. Second, Reverb allows per-user, *differential* customization of the information transmitted in the configuration channel. Such customizations do not require any intervention on the part of the middleware, but instead are accomplished implicitly as part of user requests for channel access.

The changes to a pub-sub application that Reverb monitors are:

- Creation/deletion of a channel;
- Subscription to a channel as source or sink; and
- Incorporation of any user code.

The basic mechanism of Reverb is simple. Reverb-enabled applications have a lazy-instantiated, dedicated event channel embedded in the middleware layer (the RChannel). Client applications can only subscribe to the RChannel as sinks; that is, the channel is a one-way information conduit whose sole publisher is the middleware layer. The middleware detects when any of the monitored configuration changes occur and publishes an event to the RChannel. The structure of the configuration events is available through the ECho API.

The destination(s) of the configuration updates is unknown, as in all pub-sub systems. If no sinks on the RChannel exist, no communication is performed and the update is essentially a no-op. Any application may subscribe to the RChannel and process the resulting configuration updates; processes might write them to a log file, display them on a console or “bridge” them to other logging facilities such as `syslog`.

In many cases, it is desirable to control the dissemination of configuration information. Situations where configuration changes must be verified in real-time or offline require that only authorized users have access to the RChannel.

Reverb applications operate using the D3P extensions to ECho, and so must obtain capabilities to create channels, to subscribe to channels as sources or sinks, or to incorporate user code. This provides the Reverb subsystem with two distinct opportunities to report on configuration changes - when access is requested for a particular middleware object (for example, requesting a capability which allows subscription to a channel), and when that access is actually exercised (the execution of the subscription operation using that capability).

5.2.2 Applying Reverb With Checked Access

Should information concerning system configuration actions be available to all/any users? The question is difficult to answer generally. Some applications will want to advertise their current state for the convenience of users, which could be accomplished by monitoring configuration changes. Others may want to conceal any user-provided functionality to protect proprietary interests or reduce outside vulnerability. A flexible answer to this question of policy is desirable.

Reverb, through its interaction with the Overwatch, provides a flexible mechanism for satisfying varied policy requirements. Applications provide policy statements (as XML documents) to the Overwatch, which uses them to decide on access requests. Consider an application whose operators wish its configuration changes logged to a file, but not displayed in any other way. The application would provide an XML policy statement to the Overwatch stating that only programs with a particular authentication token can subscribe to its RChannel. A legitimate logger program, possessing the correct authentication token, would then provide it to the Overwatch as part of its request to subscribe to the RChannel.

5.2.3 Customizing the RChannel

A commonly cited [25] drawback to using pub-sub architectures is the lack of control over communication suffered by the client. In the Reverb context, applications subscribing to an RChannel might find themselves spending CPU cycles and network bandwidth information they do not need or want. We address this problem through *client-specific customization* of pub-sub systems. More fully described elsewhere [52], client-specific customization allows subscribers to define filter functions that are dynamically compiled and executed at the publisher.

Reverb extends the notion of client-specific customization in the following way. Consider a large scientific application where one set of users is interested in the installation of any system extensions that provide downsampling access to color image streams but, as a matter of application policy, are not allowed to learn of other system configuration changes. Reverb can provide access to the RChannel that filters out all but the desired configuration information. Recall that the protected mode of ECho forces all operations to be performed using capabilities obtained from the Overwatch. Overwatch incorporates application policy in order to make decisions about what kind of channel access to grant and what data transformations to install as a precondition of access for particular users. In this example, the application policy statement to Overwatch identifies a certain subset of users that require this customized access and supplies metadata for a filter that performs the customization. When one of these users requests access to the RChannel, Overwatch

installs the filter on the RChannel, which in ECho creates a new, customized RChannel. Overwatch then answers the user request with a capability for the customized RChannel that can be used to subscribe as an event sink.

5.2.4 Supporting Infrastructure

As described in 4, certain pieces of supporting infrastructure are necessary in order for D3P facilities like Reverb to function. We describe Reverb’s impact on this infrastructure here.

- *Code repository.* A filter used to customize the RChannel is user- or system- specified code. This code is not necessarily incorporated into an application. Reverb can dynamically compile user-provided code for a filter, providing maximum flexibility and “user-in-the-loop” behavior. In cases where third-parties provide shared-library filter support, a code repository provides architecture-specific access to filter code. Integrity of binary-format filters is assured through the use of message digests.
- *Directory service.* Reverb makes use of the Proactive Directory Service (PDS) for two reasons. First, distributed access to an RChannel implies the need for global naming support and coordination of access to the global namespace. Second, PDS supports rights revocation services through its customizable *proactive* mode.
- *Group broker.* It would be inefficient in terms of time and resources to perform repeated identical customizations on the RChannel for different subscribers. To avoid this, the Overwatch performs broker function. When a subscriber’s access to the RChannel has been determined, Overwatch first checks to see if any compatible customizations of the RChannel already exist. If so, the subscriber is assigned to the already-existing customized RChannel.

5.3 Application Experience

We illustrate the use and functionality of Reverb by describing how a wide-area distributed application takes advantage of Reverb’s abilities. The application we use comes from the pervasive computing domain. Active Video Streams (AVS) is introduced in Chapter 2.1 and is described in more detail here.

5.3.1 Active Video Streams

We created Active Video Streams (AVS) to explore issues with deploying adaptive, high-performance data-streaming services in a distributed environment. By adaptive, we mean the ability of the application to react to changes in environmental conditions (network congestion, for example) or application-specific considerations (such as input from a human user). We also seek to provide such adaptive behavior at little or no cost in terms of run-time communication latency, bandwidth consumption, or more complicated costs associated with application re-engineering. Figure 16 shows the arrangement of the AVS application.

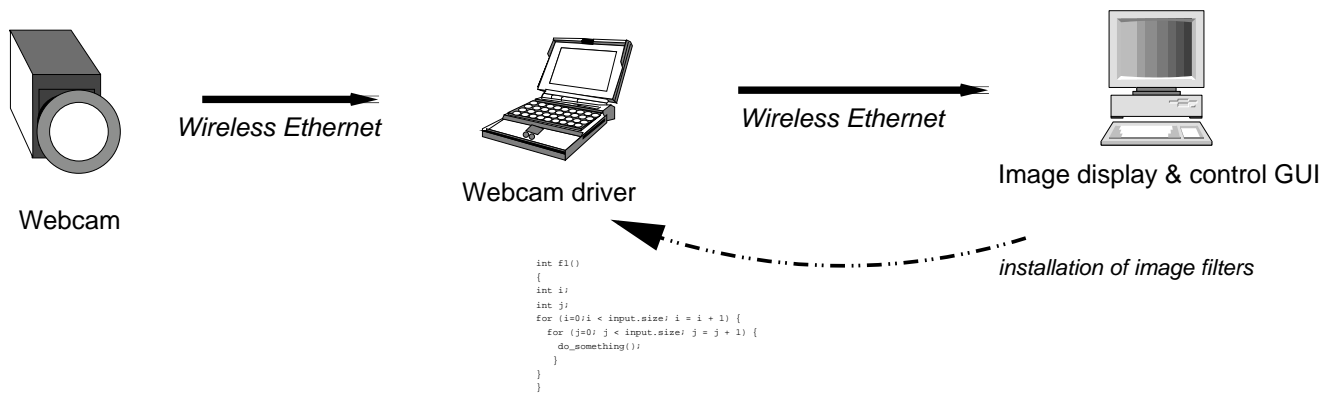


Figure 16: The Active Video Streams (AVS) applications consists of a camera driven by a separate host machine. This host machine serves as a video source and transmits images from the camera over a wireless communication link to a Java player application. The player application incorporates a control interface which can install filters on the event channel connecting the two hosts.

AVS consists of two components which communicate using the ECHO middleware. The first component is a webcam and associated driver that sends images along the communications link. An off-the-shelf webcam is used for this purpose, connected to a computer through a standard USB port. Images recorded from the camera are converted to PPM images and marshaled using the PBIO binary encoding package. Each data packet represents one frame from the video camera. The encoded data packets are then sent along an ECHO event channel to the receiver component.

At the receiving end of the event channel, the other AVS component decodes the data from its wire-format representation using PBIO. AVS has two modules that can fill the

role of the receiver component. One is a simple listener that writes frames to the local file system as they are received. The other is a Java-based viewer (the *player*) that provides the user with an interface on which each successive frame is shown. By using the viewer interface, the user effectively can see the real-time view picked up by the webcam.

An example of the type of application modification carried out in AVS is the following. The default configuration for the webcam is to send RGB images. Suppose that a particular user wishes to instead see greyscale images from this webcam. This could be due to network conditions, where the 66% reduction in data being transferred might well improve performance. It is also possible that certain applications might better serve users if the image stream could be converted to greyscale on demand. The stream modification necessary to do the greyscale conversion is a piece of E-Code that performs a mathematical transformation on each pixel of the color image to produce a greyscale image.

5.3.2 Configuration Change in AVS

The three types of configuration changes for which Reverb produces audit information correspond to the following actions in AVS:

- A camera component publishes images captured from the camera;
- A remote viewer subscribes to an existing image feed;
- A remote viewer installs a filter to change the type of image received.

For an AVS camera component to publish images, an ECho event channel must be created. The camera driver code must then subscribe to the newly created channel and begin submission of events, in this case image frames. For a remote viewer to view images produced by an AVS camera, it must subscribe to an existing image feed, i.e. the already created event channel. For the user of a remote viewer decides to manipulate the image for functionality or performance reasons, a filter must be installed on the event channel (*deriving* a new channel). Each occurrence of one of these changes in the AVS application is a significant event in the configuration of an AVS application instance. Reverb causes *configuration events* to be generated by the middleware for each case (without explicit action

by AVS or the user), and submitted to the RChannel. An interested party monitoring the RChannel can then gain a clear picture of the configuration changes in a particular instance of AVS.

We will illustrate this with a case where user code is injected into the AVS execution loop, i.e. the installation of an image filter. The corresponding ECho API for this operation is `EChannel_derive`. This API requires a reference to the original event channel and a reference to the filter code to be installed. As noted earlier, this filter code can either be provided immediately, as a text string to be dynamically compiled, or as an indirect reference to a shared object. During the `EChannel_derive` call, the Reverb subsystem notes the success or failure of the derivation attempt. Failure could occur due to an error in the code causing a problem during dynamic compilation or due to inability to locate an appropriate shared object for the machine architecture in use, among other reasons. Assuming the Reverb subsystem is configured to log derivation events, an RChannel is instantiated if need be. This involves creating an ECho event channel and subscribing as a source (actions which are themselves logged by Reverb). The configuration event is then submitted to the RChannel.

The configuration event contains a type indication (`CREATION`, `SUBSCRIPTION`, or `DERIVATION`) and reflects whether or not the operation succeeded. An application/component identifier can be provided to Reverb; if so, it is included in the configuration event and can be used to uniquely identify the source of component messages. Finally, operation-specific information (such as the code string used to derive a new ECho event channel and the channel identifier of the source channel) is also included.

5.3.3 Access to the RChannel

From the subscriber point of view, the RChannel is just like any other ECho event channel. ECho supports global identifiers for its event channels but has no namespace or discovery infrastructure. AVS retrieves and publishes the global identifier for its RChannel in a directory service with well-known contact points. Any application wishing to subscribe to the RChannel can then retrieve this global identifier and subscribe normally.

This situation changes if checked access is being performed through the Overwatch security module. In this case, the ECho middleware notifies Overwatch of the RChannel identifier. Anyone wishing subscription or derivation access to the RChannel must provide appropriate authentication credentials to Overwatch. The nature and content of these credentials can be specified by applications in a policy file. For instance, applications may specify that username/password authentication against `/etc/passwd` is sufficient for RChannel access, or instead that public-key certificates must be provided. AVS currently identifies users through username/password pairs. The global identifier for the RChannel is not provided directly in checked access mode either; a capability is issued to the requester that can be used to subscribe to the RChannel through the checked access API of ECho. This provides a protected interface to configuration events, one of Reverb's notable contributions.

5.3.4 Customization of the RChannel

For applications that undergo frequent configuration changes, a significant number of configuration events may be generated. In the AVS context, this can occur when several users introduce individual customizations into the event channel, each operation deriving a new channel. Since the RChannel is a normal ECho event channel, it too may be customized by the introduction of user code.

A basic example is of a monitoring application that is only interested in AVS derivation events. Applications can provide a filter that inspects each event, checks its type field, and discards it if the type field is not `DERIVATION`. Since the structure of configuration events is available through the ECho interface definition, user code can be written to enforce any desired filtering action. Also possible is synthesis, where a filter may compute data for its own later use or to be provided to downstream parts of notification structures.

As with subscription, the situation changes in checked access mode. The convention for derivation in ECho in checked access mode is to provide authentication credentials and a reference to user code to ECho, which in turn provides them to Overwatch. A capability for

a newly derived channel is then returned to the caller. More frequently, however, applications will define *a priori* what access to configuration information particular users (or user classes) will have. This is dictated by the principle of *least privilege*, under which a user or component of a system has access to the smallest subset of system information necessary for its job. These policy definitions are made available to Overwatch and are used to respond appropriately to RChannel access requests.

Significant here is that audit processing can be subdivided and customized, without requiring intervention from the audit subscribers. Instead of a monolithic audit application receiving and de-multiplexing all AVS configuration change events, multiple smaller applications are created. Each audit application receives unique authentication credentials, and an AVS policy statement detailing which configuration events are accessible by which audit subscriber is provided to Overwatch. For example, suppose that two audit information subscribers exist for AVS, one to process information about channel creations and subscriptions (CS) and the other to process derivation information (D). This is not an unreasonable construction of an AVS application instance; an audit program which verifies that user-supplied filters are installed in a predetermined order does not necessarily need to know when subscriptions and unsubscriptions happen. When D requests access, Overwatch notes its restriction from the policy statement. Overwatch then provides D with a capability that refers to a *pre-customized* RChannel; that is, an RChannel that has already had a filter applied to it. In this case, the filter is one that checks the configuration event type field and only passes those events of type DERIVATION. D can only access RChannel information through the capability; it cannot forge a new capability to “get around” the installed filter. RChannel access for CS is provided similarly. In this way, audit components can be kept more modular and their vulnerability is reduced. Similar checked access to the RChannel can be implemented for audit code that synthesizes events.

5.3.5 Response to Changes

What do applications do with the information provided through an RChannel? Part of the design philosophy of Reverb has been to separate these considerations from the RChannel

mechanism. The question is mainly one of policy: what action should be taken in a particular circumstance? The Reverb approach to policy is to incorporate such decisions in the Overwatch module, and to have applications provide corresponding policy statements when they begin execution. In AVS, for example, if a customization is installed that corrupts or otherwise disables an ECho channel, audit components can provide feedback to Overwatch. Such policy updates might include disabling customization for a particular end user or restricting the use of suspect customizations for all users. It seems natural to consider the pub/sub paradigm as a method of managing such policy updates. Overwatch does not currently support this type of adaptive policy management. However, our ongoing research is considering how best to address such issues, and how to better shift the design space from a “solid” policy/mechanism boundary into sets of cooperating components.

5.4 Reverb Summary

This chapter has described the design and implementation of Reverb, a product of a middle-ware implementing D3P principles. Reverb is designed to support efficient, customizable, and protected dissemination of configuration information for distributed, large-data applications.

Our implementation of Reverb illustrates the flexibility and possibilities of using D3P. Developers may prefer a maximally preventive system, defining access control policies for use by Overwatch and disallowing all other access. A completely permissive system with detailed logging output is also possible; in this scenario, permissive policies at Overwatch coupled with detailed Reverb auditing output give the desired result. Given appropriate policy definitions, options in the design space between these two extremes can be explored; for example, particular user groups may be subject to increased auditing while specific applications may desire complete access control. These policy definitions are orthogonal to each other, allowing application security administrators and functionality developers to work independently.

In Chapter 6, we shown how Reverb utilizes the D3P performance philosophy: rich functionality (in this case, detailed and customizable audit trails) for our target applications

is possible while imposing small and amortized performance costs.

CHAPTER 6

EVALUATION

This chapter presents evaluation results that quantify the costs and benefits of the D3P design approach. These results have been obtained by experimentation on the reference implementation of D3P, which is a modification of a snapshot of the ECho and PBIO libraries. We first present a short discussion of the performance philosophy that drives the design of D3P, along with a characterization of basic overheads incurred by the implementation. As the performance of ECho itself has been characterized elsewhere, we will confine our discussion to the impact of the D3P modifications to ECho.

In Section 6.3, we demonstrate how our implementation of D3P scales along the dimensions of number of subscribers and sizes of data. D3P is designed to support large user populations as well as large data applications, so acceptable performance under increasing subscriber loads and data sizes is critical. We also discuss measurements derived from the Reverb audit facility of D3P. Reverb provides applications with a “lazy” alternative to the D3P approach, and the performance of those applications during periods of frequent configuration activity can be a factor in determining whether or not to use it.

In Section 6.4, we present an evaluation of the XMIT metadata tool. Much of the flexibility benefits of D3P depend on the ability to represent type information in a system-agnostic manner, as D3P does with its XML Schema-derived method. We present experimental results that quantify the implications of representing type structures in this manner.

Section 6.5 demonstrates performance benefits available through the use of D3P. Using D3P, applications can meet functional and/or security requirements without sacrificing efficiency. In some important and common cases, performance is actually improved over alternative, requirement-preserving implementations.

The chapter closes with a summary of our evaluation results.

6.1 Data Path vs. Control Path

The key insight to the performance benefits of D3P is the realization that many performance-limiting operations in the applications happen in what can be termed the *control path*. In the streaming data applications addressed by D3P, the control path encompasses stream definition, changes to the type of data being streamed, or installation of handlers on the stream. The amount of time taken to perform these operations is typically dominated by the time taken during normal streaming. For example, consider the AVS application. A user may view a stream of several thousand images, decide to install one of a set of handlers, and then view another set of image frames. In this example, time spent receiving images constitutes time spent in the *data path* of the application. The performance implications of D3P are driven by the degree to which the time spent in the data path dominates the time spent in the control path.

We now consider the control exerted by the D3P system. By definition, the actions governed by D3P are in the control path of an application. Policy statements determine whether users can subscribe to streams, whether they can install handlers, and which handlers they can install. Once the subscription has been completed or a handler has been installed, data transfer continues at rates largely unaffected by protection actions.

D3P does introduce some performance overhead into the data path for each event received. The capability supplied by the user must be checked to see that its rights have not been changed or revoked entirely. This involves computing a hash to verify capability integrity, checking a revocation flag and performing a bitmasking operation. These steps are taken for each subscription held by a receiving process. However, they are only necessary once per event for a sending process, no matter how many subscribers exist. This reduces the ability of subscribers to perturb the performance of publishers.

6.2 Basic D3P Overhead

We first characterize the overhead of our D3P implementation in ECho, by measuring the time taken to create basic ECho objects with and without D3P. For each middleware action, we record the percentage increase in time required to complete the action. The following

table presents representative results from these tests.

ECho operation	percentage overhead
channel create	4.52
channel subscribe	3.32
handler install	8.55
handler uninstall	3.33

The primary attractiveness of the D3P mechanism is that these performance overheads are not in the critical path of data transfer. In the reference implementation, for instance, once access to the channel has been established or a handler installed, data transfer proceeds at speeds limited only by the underlying middleware (ECho) or network. Even in those situations where the mechanism does have a performance impact, that impact is minimal.

6.3 Scalability

In this series of experiments, we show that the addition of D3P functionality does not impact the scalability of the ECho middleware. For each experimental scenario, we report several metrics. Total send time is measured as the “wall-clock” time needed by the server to send a set of events to an increasing number 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.

We have previously demonstrated [24] that ECho is linearly scalable with respect to the number of subscribers and the amount of data transferred, even though it is essentially a unicast architecture. Figure 17 depicts this result. Our insight into the design of D3P is that this efficiency can be preserved through careful attention to the data transfer path.

We repeat our previous experiment using the D3P version of ECho. Results appear in Figure 18. As expected, absolute performance is worse than the unprotected ECho case.

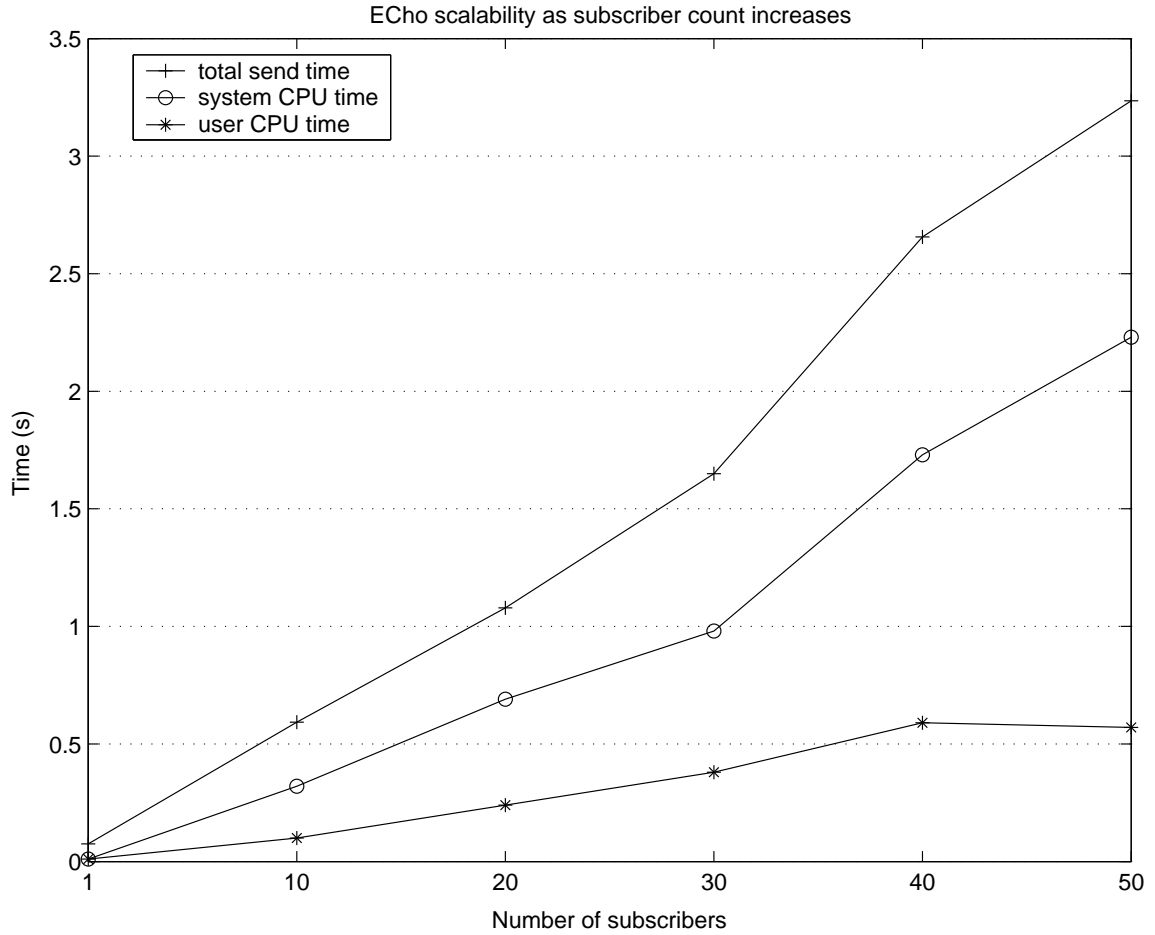


Figure 17: ECho scalability with no D3P features.

However, performance still degrades linearly with the number of clients on a channel. In particular, the degradation is less noticeable (more fully amortized) as the number of subscribers increases. This is attributable to the fact that the added D3P protection operations (verification of capability integrity and rights) have most of their impact outside the “fast path” of data transfer (at channel subscription time or handler installation time). In the data transfer path, a small set of overheads are incurred and they become less significant as the number of subscribers increases (more and more time is spent in non-protection middleware operations and the network stack). The primary contributors to this set of overheads are the verification of sender and receiver capability integrity (computing a hash) and the need to check a revocation status indicator. Both overheads are independent of the number

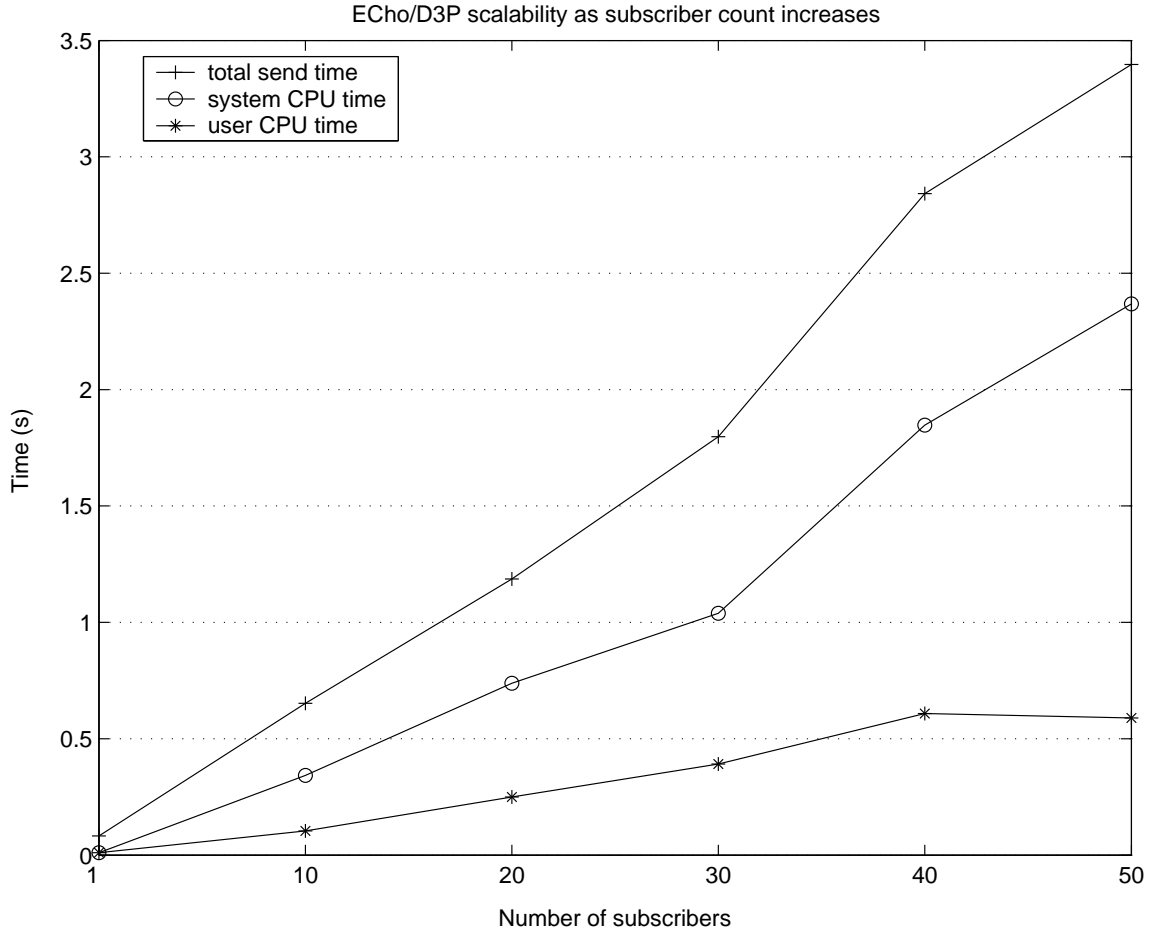


Figure 18: ECho scalability where increasing numbers of clients use the D3P interfaces.

of subscribers and are only incurred once per event submission. Their cost as a proportion of the total data transfer cost becomes less significant as the number of subscribers increases.

6.3.1 Reverb Measurements

For large-scale, widely distributed applications, many entities may be interested in configuration information. This may mean that applications have multiple D3P-managed subscriptions. As the number of subscribers grows, it is important that the performance of Reverb scales in a linear fashion. We believe it reasonable to assume that in most cases, the number of application clients will be larger than the number of clients for Reverb configuration events.

Previous research [24] has demonstrated the scalability of the ECho middleware. We

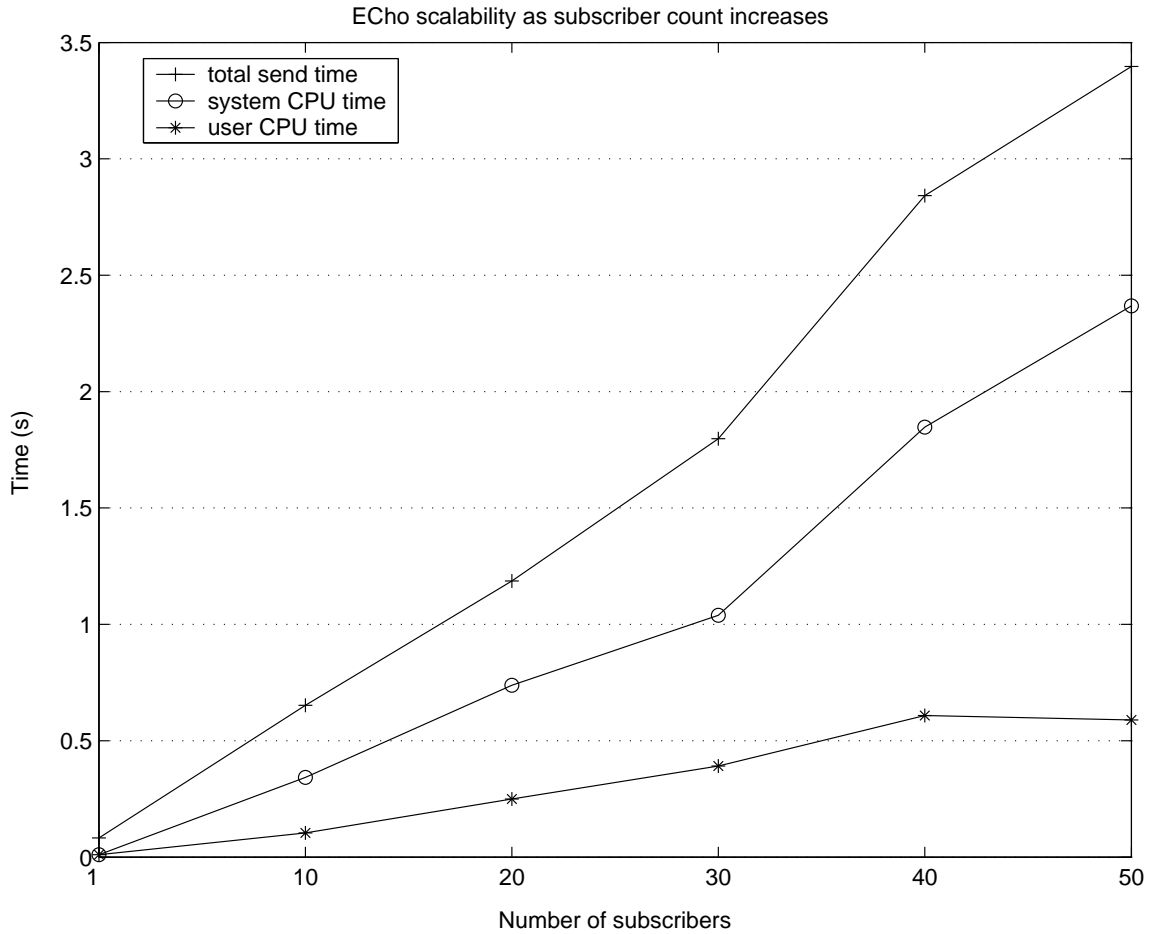


Figure 19: ECho scalability under increasing numbers of clients with a fixed set of Reverb subscribers.

executed a modified experiment using the checked access interfaces to ECho, with Reverb configuration events enabled, and set up 5 clients (on separate hosts) for Reverb configuration events. Quantities measured were the amount of CPU and wall-clock time consumed in performing a scripted series of AVS actions (which resulted in a set of configuration changes for which audit information is generated along the RChannel), during which a fixed number of image frames were received. As shown in Figure 18, performance degrades roughly linearly with the number of clients on a channel. In particular, the degradation is less noticeable (more fully amortized) as the number of subscribers increases. This is attributable to the fact that Reverb communication is increasingly dominated by the “data path” as the number of image subscribers increases.

6.3.2 Customization

As described earlier, a key feature of Reverb is the ability for subscribers to get information *differentially*; that is, they can tune the Reverb data stream to receive exactly the proportion of events they are interested in. Previous research [52] details the benefits of discarding traffic at the source instead of the sink, and Reverb channel customization leverages this concept.

This experiment repeatedly increased the number of subscribers of the RChannel, dividing them into halves which discard differing proportions of the channel traffic. For each amount of subscribers, a scripted series of AVS actions, designed to produce a number of update events, was repeated. Quantities measured were CPU and wall-clock time consumed in RChannel operations. In the first set of tests, half of the clients discard 20% of Reverb traffic and the other half discards 80%. The second set of tests changes these proportions to 50% and 80%, respectively. These results are available in Figures 20 and 21.

These results illustrate that the performance of the D3P/Reverb system scales in a linear fashion even as customization actions are being performed. This is significant because it is highly likely that Reverb clients will only be interested in a subset of configuration events, and therefore that customizations will be installed on the RChannel. Furthermore, it is also likely that clients will be interested in different subsets of events, and these results illustrate linear scalability even under such conditions.

6.4 XMIT Evaluation

D3P uses the XMIT metadata toolkit to represent structured type definitions in a manner independent of any particular middleware library. This approach provides usability benefits to applications which can use XMIT's XML Schema-derived syntax for defining type structures. As D3P relies heavily on XML documents that specify handler and policy metadata, and also on conversions between user-specified types to enforce protection decisions, XMIT is a natural choice for D3P.

We have constructed a set of experiments and used them to evaluate XMIT's performance. Our results indicate that using XML as a metadata definition language can yield a

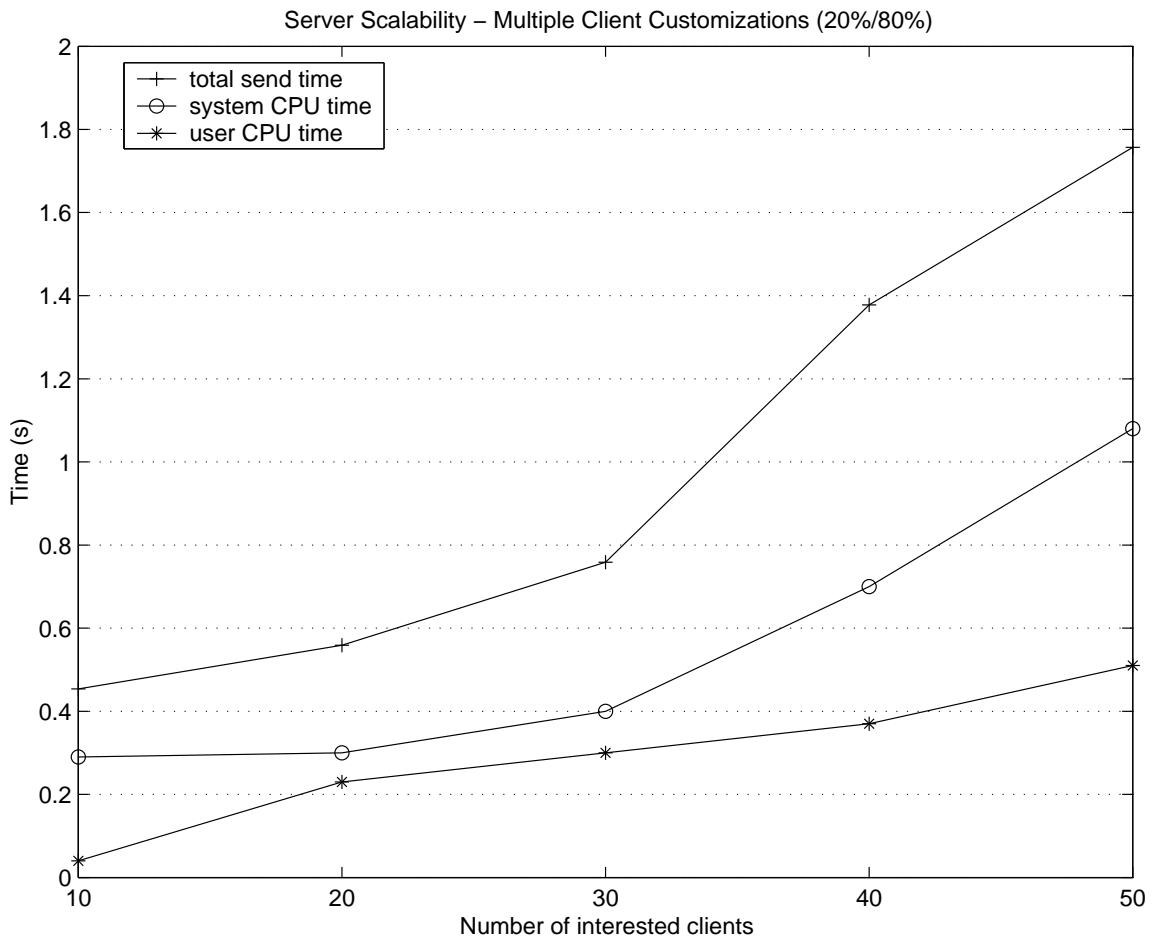


Figure 20: Server scalability where half of the clients discard 20% of Reverb traffic and half discard 80%.

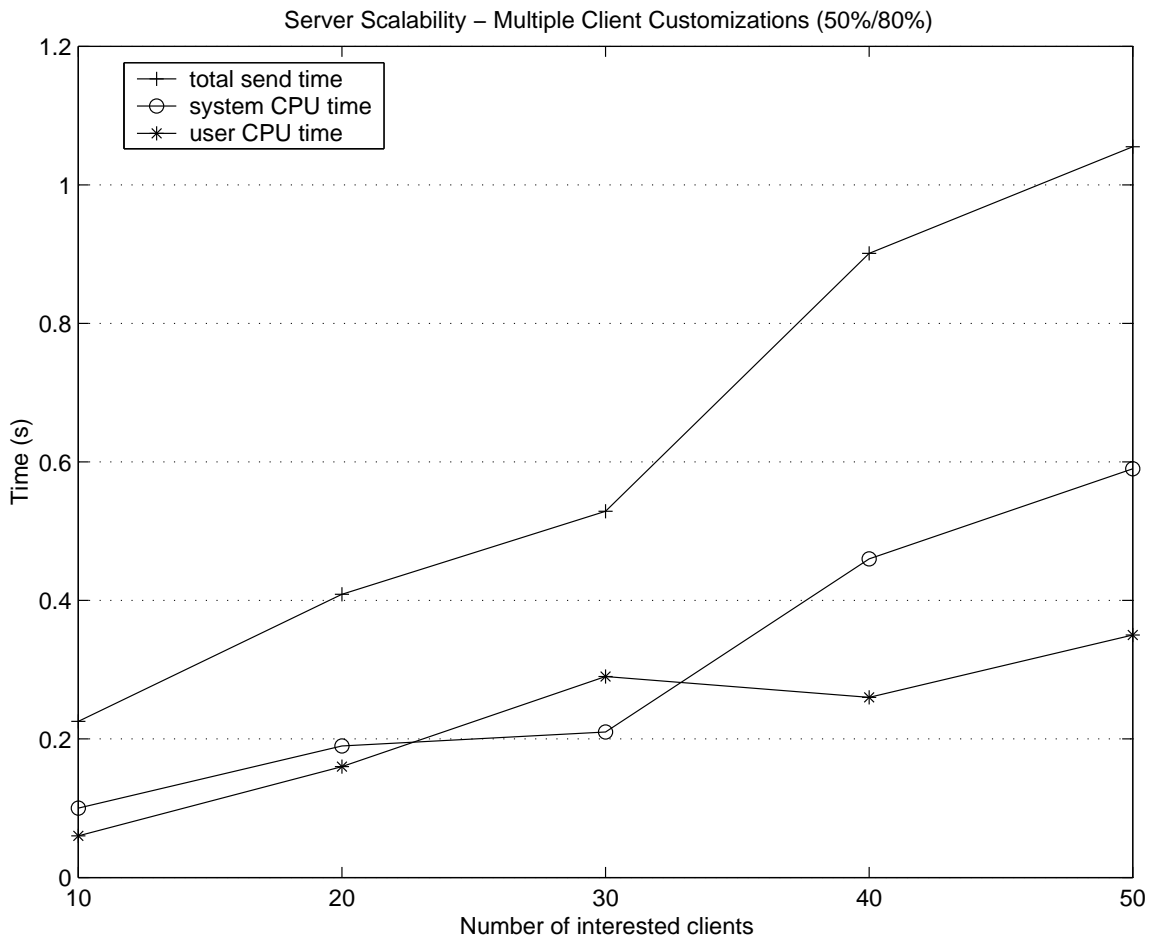


Figure 21: Server scalability where half of the clients discard 50% of Reverb traffic and half discard 80%.

large gain in usability with almost no performance penalty:

- Binary transmission of structured messages is superior in almost every case to using XML as a wire format. This holds *except* when senders and receivers use XML directly, when they transmit messages as unstructured ASCII, and when these messages are small. In this case, the additional overheads implied by the use of XMIT outweigh the benefits attained from the improved efficiency of binary transport.
- For the high performance applications and usage scenarios for which XMIT was developed (e.g., for D3P applications such as AVS), we have experienced superior performance for binary transport using XMIT even when binary transport performs in its worst case (small messages, encoding/decoding required at both ends) and XML transport is at its best (small messages, no encoding/decoding required). In one application-based experiment, XML messages are 3 times larger than the corresponding binary messages (see Figure 22 for the structure definition and XML encoding), resulting in the XML-based solutions experiencing twice the latency than the solutions using XMIT.

```
typedef struct
{
    int height;
    int width;
    char *data;
} ImageData

<ImageData>
  <Height>640</Height>
  <Width>480</Width>
  <Data>12</Data>
  <Data>33</Data>
  <Data>55</Data>
  ....
  ....
  ....
  <Data>00</Data>
</ImageData>
```

Figure 22: A C structure defining a sample structure representing an image, and a sample XML encoding of the structure. The XML expansion results in a considerably larger representation of the data, significant when exchanging many messages.

The introduction of XML-based metadata into D3P, whose communication is performed using PBIO for wire-formatting, adds no additional overhead to data transport. For this reason, it is not meaningful to compare communications times of applications with and without such metadata. Any metadata-related overhead occurs only at program startup; since the approach we describe uses PBIO format registrations derived from the provided XML format descriptions, PBIO-based communications can continue as if normal PBIO metadata were being used. This allows any increased cost of discovery and registration to be amortized across the entire set of messages sent using a particular metadata format. As the number of messages sent in a particular format can reasonably be expected to dominate the number of format discoveries and changes, the overall effect on performance should be tolerable. Note also that in both compiled-metadata and IDL-metadata systems, format changes require manual intervention at every source and sink point (in the form of recompilation of systems that cannot cope with the format changes).

The impact of using XMIT lies in the additional time required to retrieve XML Schema-based metadata and the time required to parse and construct PBIO metadata. We define the **Remote Discovery Multiplier** (RDM) as the ratio of the time needed by XMIT to register a message format with respect to the time needed by PBIO to register the same format using compiled-in metadata. RDM provides an indication of the *cost of remote metadata*; that is, the quantifiable performance penalty associated with the harder-to-quantify usability benefits derived from the use of XMIT.

The gains in usability attained from the use of XML for data definition imply only small additional costs compared to using lower-level metadata representations like PBIO. In Figure 23 we characterize the time required to parse and register metadata for different structure sizes. Format registration time for XMIT includes the time necessary to parse the XML description of the format and register the format with PBIO. **Structure Size** is the size of the language-level structure in bytes. **Encoded Size** represents the size of a buffer resulting from a marshal operation in PBIO for each metadata approach.

Note that the Remote Discovery Multiplier remains relatively constant even as the structure size increases. This indicates that XMIT imposes a constant factor of overhead

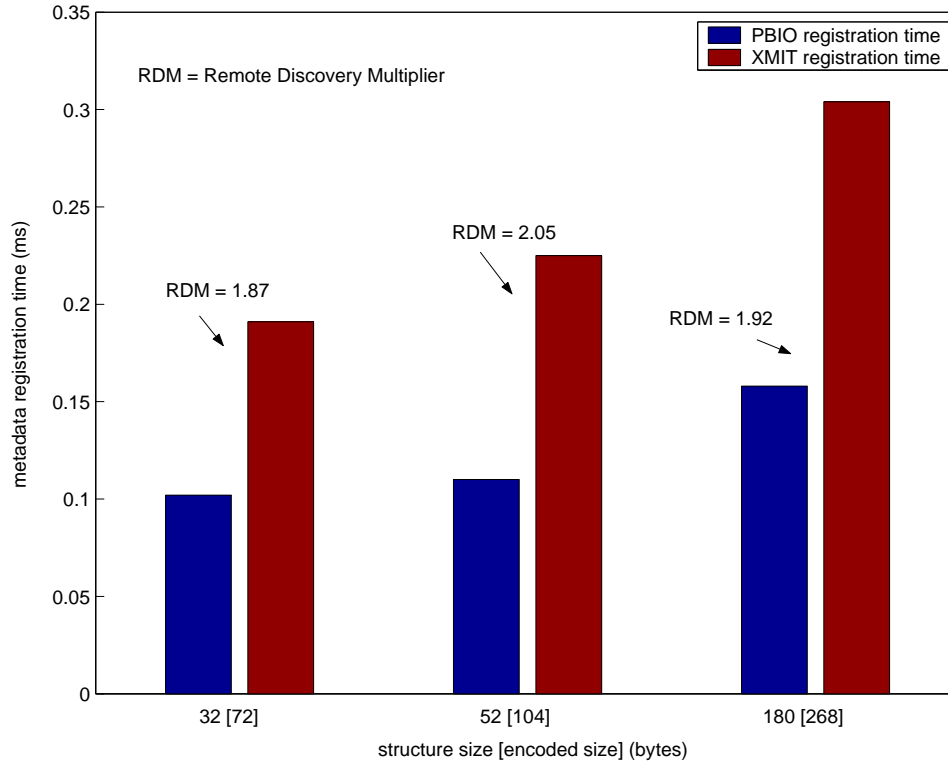


Figure 23: Format registration costs using PBIO and XMIT.

on the metadata discovery process. Also, it is important to note that registration time does not necessarily increase in strict proportion to message size, but instead corresponds more closely to the complexity of the message (in terms of size, number of fields, and nested definitions). For the sample formats used in these experiments, complexity and size are related linearly (i.e., the larger the size, the more fields). Additional structure definitions and equivalent XML Schema representations can be found in [172]. Finally, the structure definitions for this experiment were retrieved from a local file system. Additional network latency for HTTP retrieval of the document would increase the absolute time required for discovery and retrieval by some time proportional to the size of the XML document. The measurements in this figure show that the additional flexibility attained by use of XML comes at a small price. The performance penalties implied by using XMIT depend on the complexity of the data structure being used.

6.4.1 Impact on Application Size

XMIT is a run-time library that must be linked into an application. The most significant cost associated with this fact is that XMIT contains an XML parser library. The memory used by the parser library is much larger than that used by natively compiled metadata. The parser used by XMIT, `libxml2`, when compiled unoptimized in our test environment with GCC 3.3, is approximately 1Mb. For smaller applications that wish to use binary data transport, this footprint may be unacceptably large. Smaller XML parser packages exist; Expat[33], for example, when compiled in our environment, is just under 290Kb. Expat does not provide the DOM functionality used by XMIT, but modifying XMIT to work with a smaller XML parser would be straightforward.

6.4.2 XMIT Summary

Our experiments support our contention that it is possible to obtain the usability benefits of XML-based metadata without a significant loss in performance with respect to native binary-format metadata. Furthermore, we have shown that XML-only data transmission, while providing the same usability benefits, does not allow the high-performance communication that a distributed application using XMIT can achieve. Applications using XML-based metadata, such as D3P applications using XML for policy and handler metadata definition, consequently do not incur unreasonable performance overheads due to the use of XMIT.

6.5 Performance Improvement

This section shows that the performance of a typical pervasive application does not suffer when using D3P. We measure the time required to perform 10000 typical AVS image exchanges of 921600 bytes (640x480 color), 230400 bytes (320x240 color), and 57600 bytes (160x120 color), with and without D3P features enabled. Figure 24 shows that, as data size increases, the sending time increases in a roughly linear manner. Also, the difference in sending time is smallest with the largest event size, reinforcing our contention that D3P costs are outside the critical data path and therefore can be amortized across larger events.

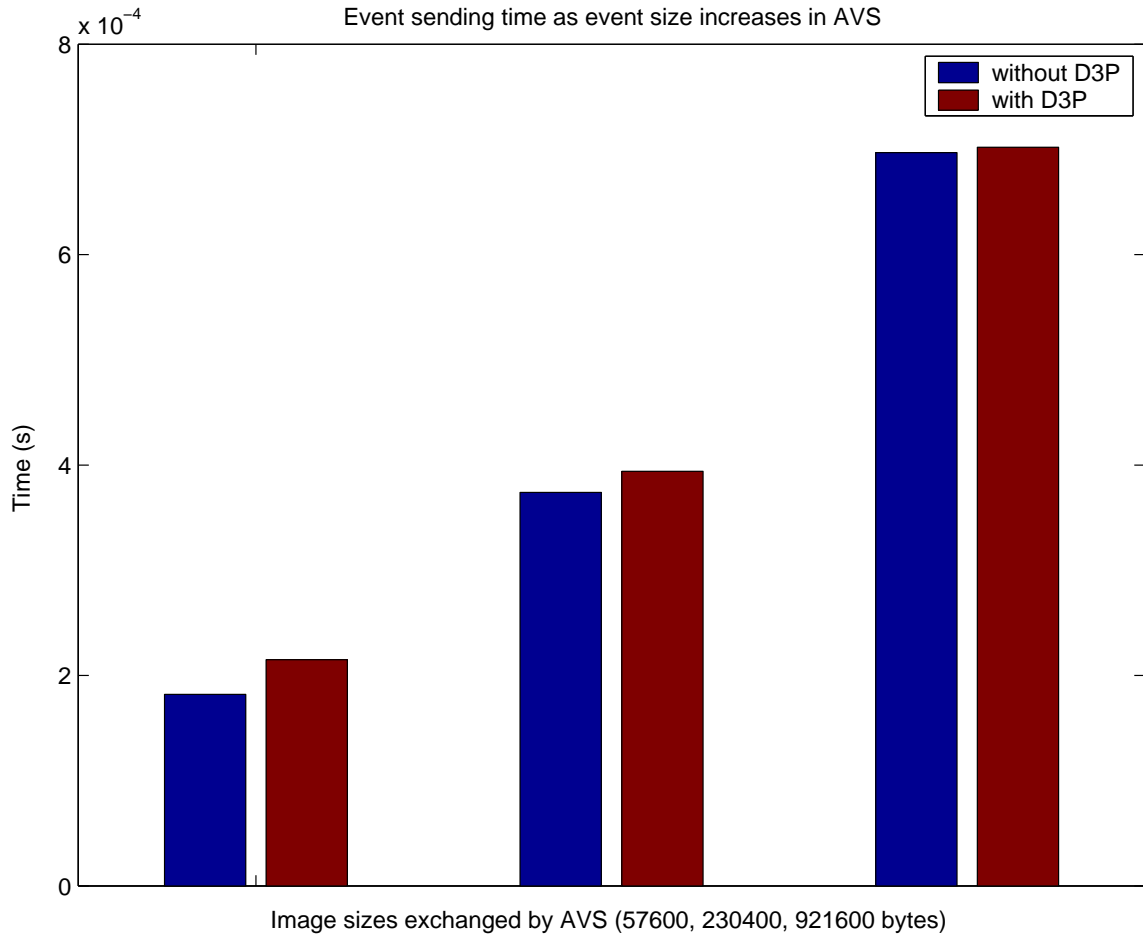


Figure 24: ECho/D3P scalability as event sizes increase.

Finally, it is worth noting that at times the use of D3P can improve performance in applications like AVS. Consider a situation where an AVS-like application without access to D3P functionality operates under a security policy where certain clients are only to receive greyscale images. If a separate image stream is undesirable, full-color images must be transmitted to and converted at the client. D3P avoids this by performing stream modification at the image source and transmitting reduced-size data to exactly those clients specified by the policy.

We analyze this situation as follows. Assume an image transfer application similar to AVS, with a single image source. If PPM images are used, the size of a greyscale image is 33% of the size of a color image. We consider an exchange of E images distributed via

unicast to N viewers, where D is the amount of time required to transmit a color image and $D/3$ the time required to transmit a greyscale image. In the non-D3P application, the times T_{server} and T_{client} required for this exchange at the server and client, respectively, are

$$T_{server} = setup + E \times (N \times D)$$

$$T_{client} = setup + E \times (D + F)$$

where F is the time required to execute a function at the client which modifies the image from color to greyscale. For the D3P application,

$$T_{server} = setup + E \times (P_{event} + F + (N \times D/3))$$

$$T_{client} = setup + E \times (P_{event} + D/3)$$

P_{event} is the per-event overhead incurred to carry out D3P-related operations; as noted above, these include computing a hash function and checking a status flag. F is again the time required to execute a function that converts the color image to greyscale, and there are two important observations here. The first is that we can consider this term roughly equivalent to the time F required to run a similar conversion function in the non-D3P case (due to D3P's use of dynamically-compiled handler functions). The second is that the function need only be executed once, at the server, in the D3P case, not once per client as might be assumed.

The *setup* term in each expression above breaks down into event channel creation, subscription, and handler installation components. As noted above, D3P overheads compared to unmodified middleware are relatively small, in the 4-9% range. Important to note here is that the Overwatch module provides the appropriate channel capability to each D3P channel client *with the handler pre-installed*. Although D3P setup costs are slightly higher, only a single handler installation overhead is necessary in the D3P case, and all setup costs happen outside the data path.

This analysis pits the overhead of D3P against the performance savings realized from transmitting only the required data. In this instance, a non-D3P application complying with the required security policy would transmit almost a megabyte of data per client, only

to discard 2/3 of it. A D3P solution provides policy compliance with overheads outside the data path and improves application performance by only sending exactly the data each client requires.

6.6 Evaluation Summary

This chapter has presented performance evaluations of the D3P programming approach, as characterized by experimentation using the reference implementation as a part of the ECho middleware library. Our performance contentions for D3P rest on the principle of applying D3P actions, as much as possible, in the *control path* of applications. This principle frees up the *data path* of applications to execute at speeds limited by the middleware and/or network. We have demonstrated that the overheads imposed by D3P on the control path are small relative to their corresponding operations in ECho. We have also shown that in the data path of applications, where actual application data is being transferred, D3P-enabled applications preserve the high-performance and scalability properties of the underlying middleware. The performance benefits of using D3P has also been shown, enabling applications in certain situations to satisfy complex security or functionality requirements with correspondingly lower performance impact than alternative solutions would require. We conclude with an evaluation of an important D3P supporting component, the XMIT metadata toolkit, that demonstrates the efficiency and practicality of representing type information in an abstract manner.

CHAPTER 7

RELATED WORK

This chapter reviews a selection of research efforts not elsewhere mentioned in the dissertation that share design goals and principles with D3P. Our aim is to place D3P in a broader research context and illustrate some of the features that make D3P unique. We present related research from domains including distributed computing, software engineering, and security and protection.

7.1 Trusted Computing

Trusted computing has historically been an area of active research, and is now enjoying renewed attention from both academia and industry. D3P addresses application requirements that are important to understand in the context of these ongoing research projects.

Definitions of trusted platforms [157, 36] promise to incorporate protection decisions into hardware, realizing an architectural ambition that has existed since Multics. Shapiro et al. [145] have revisited the idea of making capabilities a first-class operating system concept, using aggressive caching to offset the lack of custom hardware support. Current research is converging on differing varieties of virtualization. Hypervisors such as Xen [50] support guest operating systems with a virtualization layer atop hardware and devices. Virtual machine monitors [68] export custom virtual machine definitions corresponding to application-defined trust requirements.

7.2 User-level Protection Mechanisms

[49] presents a scheme for implementing anonymous group signatures which is similar to the signature scheme used in D3P to secure capabilities. Their method also utilizes a Mediator process that is separate from applications and provides revocation services. The D3P Overwatch serves a similar purpose. However, Overwatch also incorporates application

policy into its decision-making, something not addressed by the Mediator. A version of this technique could be successfully integrated into D3P’s capability system.

As stated earlier, the threat model addressed by D3P explicitly does not include subversion of local address spaces. This choice was made largely because of other current research into this problem. Several projects have proposed different methods of protecting address spaces without resorting to kernel-level functionality, of which future D3P implementations might make use.

Address obfuscation [17] randomizes the locations of user code and data on every execution (via customized linker/loader activity) in order to defeat exploits that depend on deterministic address allocation. Such a system could be integrated into the D3P handler execution system in order to defeat potential malicious address inspection. TaintBochs [31] examines heap-allocated data lifetimes, a potentially useful simulation tool for D3P channels that might use persistent data and exist over relatively long time periods. Systems like PointGuard [39] protect against buffer overflows by obfuscating or encrypting address references and resolving them only when needed.

Sandbox-style approaches [71, 163, 164, 127] can also enforce user-level protection by examining instructions on execution to ensure no policy constraints are violated. Program shepherding [105] monitors control-flow transfers to ensure that only desired code is executed. D3P provides equivalent functionality to these approaches, restricting unsafe handler operations through language design and ensuring via Overwatch that appropriate, verified handlers are distributed based on policy. Another execution-oriented tool is Systrace [130], which uses an interposition technique to enforce application-specific security policy on system calls.

Interestingly, the differential nature of data protection in D3P is approximated in the on-line analytic processing domain by work reported in [165]. Methods for defining authorization constraints on a multi-dimensional data cube (corresponding to an arbitrary data structure in the D3P world) are presented, and are notable here for their multi-dimensional approach. Arbitrary data selections are possible, and potential information inferences from unprotected data are also detected. D3P provides similar functionality through its ability

to associate handlers that can convert data from one type to another, masking data from which inferences might be drawn. The operating system-level, policy-driven and type-based protection system used in Security-Enhanced Linux is described in [97].

7.3 Security Policy Implementation and Management

D3P's Overwatch module has the task of interpreting arbitrary policy statements and producing decisions about which handlers to install on which data streams. This requires exact data type information both at the policy interpretation level (brokering) and at the implementation level (for protection services). In [112], this is referred to as a *crisp* data model. They propose the use of fuzzy set theory to process uncertainties in subscriptions. Where a D3P policy statement might read "a camera must not be aimed between 0 and 120 degrees", a fuzzy policy statement could read "the camera is aimed at the location of a warehouse". Such statements allow the expression of *gradual set membership* and allow expression of subjective (and therefore more natural and expressive) subscription or policy criteria.

The Placeless Documents System [14] embeds policy statements with access control requests, thereby avoiding the problem of capability revocation. Implemented over an SSL-equipped version of Java RMI, it decentralizes access control decisions that D3P delegates to Overwatch. Placeless is not suitable for high-performance applications due to the limitations of RMI, whereas D3P is designed expressly for those situations.

A concern for any system that centralizes policy management in a single module (such as D3P's Overwatch) is the collection, over time, of sensitive information required to evaluate the policy. For instance, Overwatch might require user location information in order to evaluate whether a capability for a particular channel should be granted. In [79], a system is proposed that uses a distributed privacy algorithm to hide locations in a sensor network. While their work only addresses location, similar concerns over other user attributes are likely to pose problems for future policy designers, and by extension future Overwatch designers.

In [82], a detailed access control design for pervasive systems is presented that closely

tracks our intended usage of Overwatch. Their design also incorporates elements of data protection or transformation between access control elements (obfuscating or generalizing user location information, for example) that is similar to the data protection actions accomplished by D3P, as well as emphasis on application-semantic (referred to as “information-level”) policy definition. Decoupling of access control policy and mechanism is also covered in their work. D3P is complementary to this effort, with the advantage of a concrete implementation of mechanism in high-performance middleware. Their design presents interesting opportunities for future research in the D3P context. The Confab system [85] addresses similar privacy concerns in ubiquitous and pervasive applications. For Grid applications, the SESAME [180] system uses existing Grid security technology to implement context-dependent authentication under a dynamic RBAC access control model.

Delegation of privileges is not currently addressed by D3P, but the design of Overwatch allows the construction of policy to realize it. If D3P capabilities are extended with appropriate data, approaches that allow them to carry references to externally-defined delegation constraints [15] are possible. In [101], an interesting approach to incorporating foreign principals into an access control matrix is proposed. Implemented in a global file system, users can register identity information with local authentication authorities; these identities can then be used in access control decisions without the need to contact a central authority (as is necessary to follow a certificate chain). D3P is compatible with this approach, allowing arbitrary identification information to be embedded into capabilities.

D3P can be used to implement higher-level application frameworks used for specific application types. A similar system that addresses computer-supported collaborative work (CSCW) applications is presented in [156]. It derives specific workflow architecture and event-subscriptions for components from a functional policy specification. Role-based access control is used to determine which workflow components have access to which information flows. Unlike D3P, their work does not support differential data access to objects. Also, their Java implementation is unsuitable for the data-intensive, high-performance application domains targeted by D3P. The DisCo [61] system is another Java-based middleware

supporting authorized access to application components. DisCo employs an Overwatch-like monitor process (*Authorizer*) and claims to address rights revocation concerns through component references, although exactly how this is accomplished is unclear. A policy-driven middleware framework developed explicitly for publish/subscribe applications is described in [118], and a system providing D3P-style protection and policy implementation for foreign executable content (although not data protection) is presented in [96]

In [65], Furst et al. provide a look at how enterprise-level access control might be realized in a Java RMI/RBAC environment. Their main issue is to consider the widely-distributed nature of enterprise data and how it intersects with policy role definition. The combination of appropriate extensions to Overwatch (currently being explored as further work) will make D3P a strong competitor in this regard, with the added benefits of high-performance and compact data representation. [37] presents a policy-definition architecture that is a useful example of the type D3P is designed to support.

The use of XML as a policy definition tool has also been a subject of previous work. [44] introduces a method of defining access restrictions on Web documents using XML. The Job Submission Description Language [100] is a XML dialect supporting brokering information of the type used by Overwatch. The Security Services Markup Language[120] proposes a method for expressing security models in XML. Additionally, although based on UML rather than on XML, QML[63] presented a method of describing system-level policies in an abstract fashion.

7.4 Protected Audit Systems

Perhaps the most widely used tool related to Reverb is the `syslog` [4] service. `syslog` provides a basic transport protocol for transmission of events across IP networks. It is intended as a lowest-common-denominator service with goals of widespread applicability and low coordination requirements. More directly related to Reverb, `syslog` provides anonymous, “fire-and-forget” event communication and a structure for definition of audit processors. In contrast to this and similar services, Reverb makes use of binary messaging as opposed to `syslog`’s text-based format, and allows arbitrary event structures to be defined. Also,

Reverb supports the introduction of user functionality into the event transmission path.

A large body of recent academic research has addressed the introduction of autonomic principles into Web Services [19, 155], middleware for pervasive and context-sensitive applications [89], and security [104]. These principles are also appearing in significant commercial offerings such as IBM's Websphere [92] and HP's OpenView [84]. Each of these efforts concerns systems that can "self-heal" or otherwise respond to configuration changes without manual intervention. Reverb is a complementary piece in this area of research, adding its high-performance and customization middleware abilities to the discussion. Also of interest are policy definition and management efforts [90] which can inform future improvements into designs of modules such as Reverb's Overwatch.

Grid services is another area where the middleware approach of Reverb can contribute. The Globus and Legion projects aim to make large sets of distributed resources manageable. Particularly, Legion research [56] and tools such as Condor [62] have begun to address questions surrounding the security of Grid schedulers and other configuration aspects of metacomputing environments. Reverb provides a flexible policy-driven approach to securing configuration information by limiting access to the RChannel. Policy management as encapsulated by the Overwatch module could be modified to explore compatibility modes with these types of Grid services.

Finally, Reverb provides access control to audit information in a manner complementary to encryption-based schemes such as that described in [142]. If forensic analysis of audit data is to be reliable, the integrity of the audit trail is paramount. Reverb addresses this issue both through access control and differential data access (which makes possible audit processing components that are smaller and therefore more amenable to inspection). [147] describes an auditing infrastructure that distributes log information among several receiving nodes so as to detect compromise of any individual node. The Reverb subsystem can be used to implement a similar system by managing appropriate subscriptions to the RChannel.

7.5 Middleware

7.5.1 Component-Based Systems

Various projects have proposed component-based approaches to software development in wide-area distributed computing [20, 81, 22]. Component architectures facilitate the construction of complex applications by allowing the creation of generic reusable components and by easing independent component development. Similar approaches have been proposed by the software engineering community over the last decade [126, 146] and their advantages have been widely recognized in industry, resulting in the development of systems such as Enterprise Java Beans [153], Microsoft's Component Object Model and its distributed extension (DCOM) [51], and the developing specification of the CORBA Component Model (CCM) in OMG's CORBA version 3.0 [122].

Targeted to Grid computing environments, CCA [22] proposes an approach to building distributed systems that is based on representing services as application-level software components. The WebFlow [81] project aims to provide a Java-based coarse grain packaging model and framework for authoring wide-area distributed applications. Blair et al. [20] propose a reflective-based approach to the design of configurable middleware together with an open and extensible component framework. In contrast with the component-based model advanced by D3P, all of these projects follow a coarse-grain, object-based approach to composition and have no provision for adaptation to changing environmental conditions or application requirements. Arguments in [78] contend that an object-oriented approach may not be best-suited to wide-area distributed computing, as it may complicate application programmability and evolution.

7.5.2 Publish/Subscribe

The publish-subscribe paradigm supported by event services is well-suited to the reactive nature of many novel applications. Publish/subscribe enables the rapid and dynamic integration of legacy software into distributed systems [123], supports software reuse, facilitates software evolution [70, 69], aids in the scalability and fault-tolerance the system and is a good fit for component-based approaches.

D3P provides communication and component integration through its use of the ECho middleware library. There exist many research efforts also targeted at the development of event notification systems and their use for component integration. Notable among these are IBM's Gryphon [151], Siena [27], Elvin [143], JEDI [41], JECho [182], and the work by Yu et al [178]. ECho is unique in its support of flexible and high performance event-based communication in heterogeneous environments. In addition, D3P/ECho enables dynamic, client-specific differentiation of application data streams through the instantiation of handler functions at channel sources.

7.6 Active and Adaptive Systems

D3P introduces functionality into the data path of applications in order to provide its unique benefits. This functionality is attached to a system component, most often the source of data, and so makes the system *active*. There is a large body of recent and mature research into such systems. Active Messages [162] are bound to user-level processing which extracts the message from the network and inserts it into an on-going computation. Several projects have extended activity to the network path [11, 18, 10] (so-called *active networks* [154]) and also to the messages within the network [95, 169]. Other introductions of activity into systems include I/O streams or disks [6, 133, 103], service definitions [72, 12], and name-to-service brokering [158].

Adaptive systems also rely on functionality in the datapath to achieve their goals. Companion research to D3P [25] has made note of the ancestry of datapath adaptivity, primitively realized in early operating systems such as Unix [134] and the Dartmouth Time-Sharing System [45]. D3P extends these primitive notions by applying data distribution and datapath adaptation to distributed applications, and by providing mechanisms to implement application-dependent policy.

Rover [98] implements a proxy-based architecture specifically tuned for client-server, mobile applications. The system uses Queued Remote Procedure Calls to overcome periods of dis-connectivity and to better utilize the network link by scheduling transactions intelligently. Rover also makes use of Relocatable Dynamic Objects to offload some resources by,

for example, trading upstream compression for network bandwidth savings. Rover has no provision for run-time adaptation to changing environmental conditions.

Odyssey [121] is another application-aware approach to adaptation intended primarily to assist client/server interaction in mobile environments. The Odyssey system consists of a *viceroys*, for resource management; a set of type-specific *wardens* that handle the intercommunication between clients and servers; and applications that negotiate with Odyssey to receive the best level of service availability. Odyssey has no consideration for the dynamic insertion and composition of wardens and provides no support for dynamic composition of adaptation across multiple nodes, making it not flexible enough to cope with the characteristics of our target environments.

The TranSend [59] proxy addresses both network and system heterogeneity by providing an extra level of indirection in the transfer paths between clients and servers. Proxies transform retrieved data, primarily images, to representation that best suit the client connectivity. In the TranSend architecture, clients rendezvous with the system through a front end. The front end contacts the load manager, which deploys transcoders on behalf of the users. Similar to the two previous projects, TranSend provides no support for dynamic composition of adaptation across multiple nodes.

Zenel and Duchamp [179] describe a general design of a proxy-based architecture that includes the notion of “filters” at an intermediate host or proxy server. While the architecture is relatively general, their system does not address issues of multiple coordinated adaptations.

DataCutter [16] is a middleware infrastructure that provides support for processing of large datasets from archival storage systems over wide-area networks. The project’s main focus is on access to archival storage data, including support for indexing and accessing of multidimensional datasets. As with D3P, an application processing structure is decomposed into a set of processes, called filters, following a stream-based programming model derived from the research group’s earlier work on active disks [6]. However, the framework is fundamentally proxy-based and does not consider run-time adaptation to variations on environmental conditions.

Proxy-based solutions have demonstrated the potential benefits of using the processing power available on the datapath, as they depart slightly from the basic client/server model by introducing a third entity, the proxy server. New environments provide additional processing units in the datapath, a potentially greater number of idle hosts and a longer, more complex network connecting clients and servers. These characteristics indicate the need for more general multi-point approach to adaptation. Although multiple proxies could be distributed over the datapath, the paradigm provides no assistance in making them cooperate.

Conductor [175] provides an application transparent adaptation framework that allows multiple adaptation-modules spread along the datapath between application and services. Conductor proposes the automatic deployment of multiple application-transparent adaptors over the datapath. Although this transparency insures backward compatibility it also limits their flexibility. In contrast to this, Active Services [12] allows client applications to explicitly start one or more services on their behalf that can transform the data they receive from end services.

Ninja [73] proposes a data flow model for composing services that is similar to the model underlying D3P. However, Ninja is intended to provide robust cluster-based services, and it does not consider dynamic adaptation of data paths or of the paths' components.

CANS [64] is an application-level framework for injecting application-level functionality into the datapath. The CANS infrastructure is closely related to D3P as both support the dynamic composition of application functionality over datapaths as well as their run-time adaptation to changing environmental conditions. CANS proposes an interesting extended-type-based composition to automate component selection based on links characteristics. The CANS infrastructure has been implemented on Windows 2000 and uses Java VM as the execution environment at its intermediate hosts. D3P and CANS differ in various important aspects: D3P focuses on wide-area, heterogeneous, and highly-dynamic environments; D3P adopts event-based techniques for component integration; and D3P explicitly targets high-performance and pervasive applications.

CHAPTER 8

CONCLUDING REMARKS

This chapter summarizes the key contributions of dynamic differential data protection presented in this dissertation. We briefly describe the design philosophy of D3P and discuss its primary benefits for applications in the high-performance and pervasive domains. We also identify future research opportunities.

8.1 Summary

This dissertation defines *dynamic differential data protection*, a novel method of providing data manipulation and protection functionality to applications. The application domains considered in this dissertation include the high-performance and pervasive areas, but D3P is also applicable in other areas such as enterprise systems. D3P provides data protection at the middleware level, and provides a mechanism that enforces application-specific policy statements. Our approach accomplishes this through the combination of three concepts: secure references to middleware objects, a data type system that supports reflection, and a model for incorporating system extensions in a protected manner.

We present case studies of how D3P directly addresses important data protection and functionality issues encountered by representative applications. D3P makes possible application designs and implementations which are vital to the successful fulfillment of requirements driven by execution conditions or application security policy. The threat models of these applications were considered and the benefits of D3P outlined. D3P does not provide memory-based protection of application objects or complete code isolation for application extensions. However, there is considerable research into these areas which can be integrated into a D3P application without difficulty. Furthermore, D3P provides protection against important classes of protection threats involving access to data streams and unchecked disclosure of application data types.

We report on the reference implementation of D3P in a mature, modern publish/subscribe middleware package. Performing this implementation automatically provides several benefits, among them being the cross-platform and standards-based infrastructure on which D3P is built. We describe how the middleware was modified in order to provide secure references to objects, to incorporate the binary wire-formatting system in use, and to provide dynamic system extension facilities. We also provide detail on the policy management component of D3P, the Overwatch module. The implementation of the Reverb middleware audit facility, based on the core of D3P, is described. Reverb provides applications with an alternative to D3P’s extremely fine-grained access control, enabling designs where configuration changes are addressed reactively (“lazily”) instead of proactively.

Finally, we report on evaluations intended to quantify the costs and benefits of the D3P approach. We present experimental results on the overheads imposed by D3P in terms of basic middleware operations such as stream creation, subscription, and extension. These results indicate that the benefits of the D3P approach are possible and result in only small overheads. We describe the core performance design philosophy of D3P. We contend that important protection actions can be accomplished outside the “data path” of applications, and therefore the cost of such actions can be amortized over increasing numbers of events and event sizes in the data path. We present experimental results that support this contention in a representative pervasive application. For commonly encountered application scenarios, application scalability using D3P remains linear, comparing favorably with scalability demonstrated by the application running on unmodified middleware. We also demonstrate that as data sizes increase, D3P-enabled middleware retains linear scalability. Finally, we provide experimental support for the performance goals of Reverb, namely that flexible and extensible application auditing facilities can be provided in an efficient manner.

8.2 Future Work

Our research into D3P has left several stones unturned. Some of these problems lie in evaluating D3P in contexts other than middleware, and others in making the original implementation of D3P more amenable to wide-area distributed system design.

The reference implementation of D3P is in publish/subscribe middleware, but the concepts are valid anywhere data protection issues exist and sufficient metadata exists. Applying D3P principles to other middleware paradigms, such as transactional and web service environments, is a promising opportunity. Another such area is at the device level, where research into providing a D3P-aware serial bus could improve data protection and bus performance for a wide range of peripheral devices. In a file system, computations could be associated with file objects, using inode contents for metadata, to implement a D3P-like solution. These efforts would do much to establish the validity of D3P as a general system design approach.

Improvements to the reference implementation are underway in various guises. The underlying middleware, ECho, has undergone significant modification and is now capable of wide-area overlay network communication. Moving the D3P reference implementation to this framework will automatically give access to a new set of applications with associated data protection problems. Pieces of the D3P architecture are also under revision, with the most promising being the incorporation of a new method of safe extension code isolation. A successful integration of this technology will allow D3P to address significant execution-time threats.

Finally, questions of application policy management should be explored in order to make D3P a more attractive system design alternative. In this area, the implementation of the Overwatch module should be reconsidered in the light of ongoing evolution of the state of the art in security policy management.

REFERENCES

- [1] “Common Object Request Broker Architecture.” <http://www.omg.org/corba2/>.
- [2] “The extensible markup language (XML).” <http://www.w3.org/TR/1998/REC-xml-19980210>.
- [3] “The infopipe toolkit.” <http://www.cc.gatech.edu/projects/infosphere/software/>.
- [4] “The BSD syslog protocol.” Internet Society RFC 3164.
- [5] ABDELZAHER, T. F. and SHIN, K. G., “Qos provisioning with qcontracts in web and multimedia servers,” in *IEEE Real-Time Systems Symposium*, (Phoenix, Arizona), December 1999.
- [6] ACHARYA, A., UYSAL, M., and SALTZ, J., “Active disks: programming model, algorithms and evaluation,” in *Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VIII)*, (San Jose, CA), pp. 81–91, October 1998.
- [7] ADJIE-WINOTO, W., SCHWARTZ, E., BALAKRISHNAN, H., and LILLEY, J., “The design and implementation of an intentional naming system,” in *Proceedings of the 17th ACM Symposium on Operating System Principles*, (Kiawah Island, NC), pp. 186–201, ACM, December 1999.
- [8] AFEWORK, A., BENYON, M., BUSTAMANTE, F. E., DEMARZO, A., FERREIRA, R., MILLER, R., SILBERMAN, M., SALTZ, J., and SUSSMAN, A., “Digital dynamic telepathology - the virtual microscope,” in *Proceedings of the AMIA Annual Fall Symposium*, August 1998.
- [9] AHAMAD, M., NEIGER, G., KOHLI, P., BURNS, J., and HUTTO, P., “Causal memory: Definitions, implementation, and programming,” *Distributed Computing*, August 1995.
- [10] ALEXANDER, D. S., ARBAUGH, W. A., HICKS, M. W., KAKKAR, P., KEROMYTIS, A. D., MOORE, J. T., GUNTER, C. A., NETTLES, S. M., and SMITH, J. M., “The SwitchWare active network architecture,” *IEEE Network Special Issue on Active and Controllable Networks*, vol. 12, pp. 29–36, May/June 1998.
- [11] ALEXANDER, D. S., SHAW, M., NETTLES, S. M., and SMITH, J. M., “Active bridging,” in *Proceedings of the ACM conference on Applications, technologies, architectures, and protocols for computer communication (SIGCOMM '97)*, (Cannes, France), pp. 101–111, September 1997.
- [12] AMIR, E., MCCANNE, S., and KATZ, R., “An active service framework and its application to real-time multimedia transcoding,” in *Proceedings of the ACM SIGCOMM '98 conference on Applications, technologies, architectures, and protocols for computer communication*, (Vancouver, BC, Canada), pp. 178–189, ACM, August 1998.

- [13] ANDERSON, R., “Why cryptosystems fail,” in *CCS '93: Proceedings of the 1st ACM conference on Computer and communications security*, (New York, NY, USA), pp. 215–227, ACM Press, 1993.
- [14] BALFANZ, D., DEAN, D., and SPREITZER, M., “A security infrastructure for distributed Java applications,” in *Proc. IEEE Symposium on Security and Privacy*, 2000.
- [15] BANDMANN, O., DAM, M., and FIROZABADI, B. S., “Constrained delegation,” in *Proc. IEEE Symposium on Security and Privacy*, 2002.
- [16] BEYNON, M., FERREIRA, R., KURC, T., SUSSMAN, A., and SALTZ, J., “Datacuter: Middleware for filtering very large scientific datasets on archival storage systems,” in *Proceedings of the 8th Goddard Conference on Mass Storage Systems and Technologies/17th IEEE Symposium on Mass Storage Systems*, (College Park, MD), pp. 119–133, March 2000.
- [17] BHATKAR, S., DUVARNEY, D. C., and SEKAR, R., “Address obfuscation: an efficient approach to combat a broad range of memory error exploits,” in *Proc. 12th USENIX Security Symposium*, (Washington, DC), August 2003.
- [18] BHATTACHARJEE, S., CALVERT, K., and ZEGURA, E. W., “An architecture for active networking,” in *Proceedings of High Performance Networking (HPN'97)*, (White Plains, NY), April 1997.
- [19] BIRMAN, K., VAN RENESSE, R., and VOGELS, W., “Adding high availability and autonomic behavior to web services,” in *ICSE '04: Proceedings of the 26th International Conference on Software Engineering*, pp. 17–26, IEEE Computer Society, 2004.
- [20] BLAIR, G. S., COULSON, G., ROBIN, P., and PAPATHOMAS, M., “An architecture for next generation middleware,” in *Proc. IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware'98)*, 1998.
- [21] BLAZE, M., FEIGENBAUM, J., IOANNIDIS, J., and KEROMYTIS, A., “The role of trust management in distributed systems security,” in *Secure Internet Programming: Issues in Distributed and Mobile Object Systems*, Lecture Notes in Computer Science State-of-the-Art series, pp. 185–210, Berlin: Springer-Verlag, 1999.
- [22] BRAMLEY, R., CHIU, K., DIWAN, S., GANNON, D., GOVINDARAJU, M., MUKJI, N., TEMKO, B., and YECHURI, M., “A component based services architecture for building distributed applications,” in *Proceedings of the 9th International Symposium on High Performance Distributed Computing (HPDC-9)*, (Pittsburgh, PA), August 2000.
- [23] BUSTAMANTE, F., EISENHAEUER, G., SCHWAN, K., and WIDENER, P., “Efficient wire formats for high performance computing,” in *Proceedings of Supercomputing 2000*, November 2000.
- [24] BUSTAMANTE, F., WIDENER, P., and SCHWAN, K., “Scalable directory services using proactivity,” in *Proceedings of Supercomputing 2002*, (Baltimore, MD), November 2002.
- [25] BUSTAMANTE, F. E., *The Active Streams Approach To Adaptive Distributed Applications and Services*. PhD thesis, Georgia Institute of Technology, November 2001.

- [26] CARZANIGA, A., ROSENBLUM, D. S., and WOLF, A. L., “Challenges for distributed event services: Scalability vs. expressiveness,” in *Proceedings of Engineering Distributed Objects (EDO '99), ICSE 99 Workshop*, May 1999.
- [27] CARZANIGA, A., ROSENBLUM, D. S., and WOLF, A. L., “Design and evolution of a wide-area event notification service,” *ACM Transactions on Computer Systems*, vol. 19, pp. 332–383, August 2001.
- [28] CHAMBERS, C., EGGERS, S. J., AUSLANDER, J., PHILIPSE, M., MOCK, M., and PARDYAK, P., “Automatic dynamic compilation support for event dispatching in extensible systems,” in *Proceedings of the Workshop on Compiler Support for Systems Software (WCSS'96)*, ACM, February 1996.
- [29] CHEN, W.-W., , TOWLES, H., NYLAND, L., WELCH, G., and FUCHS, H., “Toward a compelling sensation of telepresence: Demonstrating a portal to a distant (static) office,” in *Proceedings Visualization 2000* (ERTL, T., HAMANN, B., and VARSHNEY, A., eds.), pp. 327–333, 2000.
- [30] CHEN, Y., SCHWAN, K., and ROSEN, D., “Java mirrors: Building blocks for remote interaction,” in *Proceedings of the International Parallel Distributed Processing Symposium (IPDPS)*, April 2002.
- [31] CHOW, J., PFAFF, B., GARFINKEL, T., CHRISTOPHER, K., and ROSENBLUM, M., “Understanding data lifetime via whole system simulation,” in *Proc. 13th USENIX Security Symposium*, (San Diego, CA), August 2004.
- [32] CHRISTODORESCU, M. and JHA, S., “Static analysis of executables to detect malicious patterns,” in *Proceedings of the 12th USENIX Security Symposium*, (Washington, DC), pp. 169–186, USENIX, August 2003.
- [33] CLARK, J., “expat - XML parser toolkit.” <http://www.jclark.com/xml/expat.html>.
- [34] COLWELL, R. P. and OTHERS, “Performance effects of architectural complexity in the intel 432,” *ACM Transactions on Computer Systems*, vol. 6, August 1988.
- [35] CONSORTIUM, W. W. W., “Simple Object Access Protocol.” <http://www.w3.org/TR/SOAP/>.
- [36] CORPORATION, M., “Microsoft corporation next-generation secure computing base (technical FAQ).” <http://www.microsoft.com/technet/treeview/default.asp?url=/technet/secu%rity/news/NGSCB.asp>, February 2003.
- [37] COVINGTON, M. J., FOGLA, P., ZHAN, Z., and AHAMAD, M., “A context-aware security architecture for emerging applications,” in *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*, (Las Vegas, Nevada, USA), December 2002.
- [38] COWAN, C., BEATTIE, S., JOHANSEN, J., and WAGLE, P., “Pointguard: Protecting pointers from buffer overflow vulnerabilities,” in *Proceedings of the 12th USENIX Security Symposium*, (Washington, DC), pp. 91–104, USENIX, August 2003.

- [39] COWAN, C., BEATTIE, S., JOHANSEN, J., and WAGLE, P., “Pointguard: protecting pointers from buffer overflow vulnerabilities,” in *Proc. 12th USENIX Security Symposium*, (Washington, DC), August 2003.
- [40] CRUZ-NEIRA, C., SANDIN, D., and DEFANTI, T., “Surround-screen projection-based virtual reality: the design and implementation of the cave,” in *Proceedings of the SIGGRAPH 1993 Computer Graphics Conference*, 1993.
- [41] CUGOLA, G., NITTO, E. D., and FUGGETTA, A., “Exploiting an event-based infrastructure to develop complex distributed systems,” in *Proceedings of the 20th International Conference on Software Engineering (ICSE '98)*, (Kyoto, Japan), April 1998.
- [42] CZAJKOWSKI, K., FITZGERALD, S., FOSTER, I., and KESSELMAN, C., “Grid information services for distributed resource sharing,” in *Proceedings of the 10th International Symposium on High Performance Distributed Computing (HPDC-10)*, (San Francisco, CA), August 2001.
- [43] CZERWINSKI, S., ZHAO, B., HODES, T., JOSEPH, A., and KATZ, R., “An architecture for a secure service discovery service,” in *Proceedings of ACM/IEEE MOBICOM*, pp. 24–35, August 1999.
- [44] DAMIANI, E., DI VIMERCATI, S. D. C., PARABOSCHI, S., and SAMARATI, P., “A fine-grained access control system for xml documents,” *ACM Transactions on Information and System Security*, vol. 5, pp. 169–202, May 2002.
- [45] Dartmouth College, Hanover, New Hampshire, *Systems Programmers Manual for the Dartmouth Time Sharing System for the GE 635 Computer*, 1971.
- [46] D.BONEH, DING, X., TSUDIK, G., and WONG, B., “Fast revocation of security capabilities,” in *Proceedings of the 2001 USENIX Security Symposium*, USENIX, 2001.
- [47] DENNIS, J. B. and HORN, E. C. V., “Programming semantics for multiprogrammed computations,” *Commun. ACM*, vol. 9, no. 3, pp. 143–155, 1966.
- [48] DEY, A. and ABOWD, G., “The context toolkit: Aiding the development of context-aware applications,” in *Proceedings of the Workshop on Software Engineering for Wearable and Pervasive Computing*, (Limerick, Ireland), June 2000.
- [49] DING, X., TSUDIK, G., and XU, S., “Leak-free group signatures with immediate revocation,” in *Proc. 24th ICDCS*, 2004.
- [50] DRAGOVIC, B., FRASER, K., HAND, S., HARRIS, T., HO, A., PRATT, I., WARFIELD, A., BARHAM, P., and NEUGEBAUER, R., “Xen and the art of virtualization,” in *Proceedings of the ACM Symposium on Operating Systems Principles*, October 2003.
- [51] EDDON, G. and EDDON, H., *Inside Distributed COM*. Redmond, WA: Microsoft Press, 1998.
- [52] EISENHAEUER, G., BUSTAMANTE, F. E., and SCHWAN, K., “Event services in high performance systems,” *Cluster Computing: The Journal of Networks, Software Tools, and Applications*, vol. 4, pp. 243–252, July 2001.

- [53] EISENHAUER, G., BUSTAMANTE, F. E., and SCHWAN, K., “Native data representation: An efficient wire format for high performance computing,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 13, December 2002.
- [54] ENGLER, D. R., “Vcode: a retargetable, extensible, very fast dynamic code generation system,” in *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation (PLDI '96)*, May 1996.
- [55] ESLER, M., HIGHTOWER, J., ANDERSON, T., and BORRIELLO, G., “Next century challenges: Data-centric networking for invisible computing,” in *Proceedings of the 5th ACM/IEEE International Conference on Mobile Computing and Networking*, (Seattle, WA), pp. 24–35, August 1999.
- [56] FERRARI, A., KNABE, F., HUMPHREY, M., CHAPIN, S., and GRIMSHAW, A., “A flexible security system for metacomputing environments,” Tech. Rep. CS-98-36, Department of Computer Science, University of Virginia, Charlottesville, Virginia 22093, USA, December 1998.
- [57] FITZGERALD, S., FOSTER, I., KESSELMAN, C., VON LASZEWSKI, G., SMITH, W., and TUECKE, S., “A directory service for configuring high-performance distributed computation,” in *Proceedings of the 6th International Symposium on High Performance Distributed Computing (HPDC-6)*, (Portland, OR), pp. 365–375, IEEE, August 1997.
- [58] FOSTER, I. and KESSELMAN, C., “Globus: A metacomputing infrastructure toolkit,” *International Journal of Supercomputing Applications*, vol. 11, no. 2, pp. 115–128, 1997.
- [59] FOX, A., GRIBBLE, S. D., BREWER, E. A., and AMIR, E., “Adapting to network and client variability via on-demand dynamic distillation,” in *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VII)*, (Cambridge, MA), October 1996.
- [60] FREEMAN, E., HUPFER, S., and ARNOLD, K., *JavaSpaces Principles, Patterns and Practice*. Pearson Education, 1999.
- [61] FREUDENTHAL, E. and KARAMCHETI, V., “DisCo: Middleware for securely deploying decomposable services in partly trusted environments,” in *Proc. 24th International Conference on Distributed Computing Systems*, 2004.
- [62] FREY, J., TANNENBAUM, T., LIVNY, M., FOSTER, I., and TUECKE, S., “Condor-g: A computation management agent for multi-institutional grids,” *Cluster Computing*, vol. 5, pp. 237–246, July 2002.
- [63] FROLUND, S. and KOISTINEN, J., “Quality of service specification in distributed operating systems design,” in *Conference on Object-Oriented Technologies and Systems*, USENIX, April 1998.
- [64] FU, X., SHI, W., AKKERMAN, A., and KARAMCHETI, V., “CANS: Composable, adaptive network services infrastructure,” in *3rd USENIX Symposium on Internet Technologies*, (San Francisco, CA), March 2001.

- [65] FURST, K., SCHMIDT, T., and WIPPEL, G., “Managing access in extended enterprise networks,” *IEEE Internet Computing*, vol. 6, pp. 67–74, September-October 2002.
- [66] GANEV, I., EISENHAEUER, G., and SCHWAN, K., “Kernel plugins: When a vm is too much,” in *Proceedings of the Third Virtual Machine Research and Technology Symposium*, May 2004.
- [67] GANEV, I., EISENHAEUER, G., and SCHWAN, K., “Kernel plugins: When a vm is too much,” in *Proceedings of the 3rd Virtual Machine Research and Technology Symposium*, USENIX Society/ACM, May 2004.
- [68] GARFINKEL, T., ROSENBLUM, M., and BONEH, D., “Flexible OS support and applications for trusted computing,” in *hotos9*, May 2003.
- [69] GARLAN, D. and NOTKIN, D., “Formalizing design spaces: Implicit invocation mechanisms,” in *VDM’91: Formal Software Development Methods*, pp. 31–44, Springer-Verlag, LNCS 551, October 1991.
- [70] GARLAN, D. and SHAW, M., “An introduction to software architecture,” in *Advances in Software Engineering and Knowledge Engineering, Volume I* (AMBRIOLA, V. and TORTORA, G., eds.), New Jersey: World Scientific Publishing Company, 1993.
- [71] GONG, L., MUELLER, M., PRAFULLCHANDRA, H., and SCHEMERS, R., “Going beyond the sandbox: An overview of the new security architecture in the Java Development Kit 1.2,” in *USENIX Symposium on Internet Technologies and Systems*, (Monterey, CA), pp. 103–112, 1997.
- [72] GOVINDAN, R., ALAETTINOĞLU, C., and ESTRIN, D., “A framework for active distributed services,” Tech. Report 98-669, Information Science Institute, University of Southern California, Los Angeles, CA, January 1998.
- [73] GRIBBLE, S. D., WELSH, M., VON BEHREN, R., BREWER, E. A., CULLER, D., BORISOV, N., CZERWINSKI, S., GUMMADI, R., HILL, J., JOSEPH, A., KATZ, R., MAO, Z., ROSS, S., and ZHAO, B., “The Ninja architecture for robust Internet-scale systems and services,” *Special Issue of Computer Networks on Pervasive Computing*, 2000.
- [74] GRIMM, R., ANDERSON, T., and AND DAVID WETHERALL, B. B., “A system architecture for pervasive computing,” in *Proceedings of the 9th ACM SIGOPS European Workshop*, (Kolding, Denmark), pp. 177–182, September 2000.
- [75] GRIMSHAW, A., FERRARI, A., KNABE, F., and HUMPHREY, M., “Legion: An operating system for wide-area computing,” *IEEE Computer*, vol. 32, pp. 29–37, May 1999.
- [76] GROUP, O. M., *CORBA services: Common Object Services Specification*, ch. 4. OMG, 1997. <http://www.omg.org>.
- [77] GROUP, O. M., “Notification service.” <http://www.omg.org>, Document telecom/98-01-01, 1998.

- [78] GRUMM, R., DAVIS, J., HENDRICKSON, B., LEMAR, E., MACBETH, A., SWANSON, S., ANDERSON, T., BERSHAD, B., BORRIELLO, G., GRIBBLE, S., and WETHERALL, D., “Systems directions for pervasive computing,” in *Proceedings of the 8th Workshop on Hot Topics in Operating Systems (HotOS-VIII)*, (Schoss Elmau, Germany), May 2001.
- [79] GRUTESER, M., SCHELLE, G., JAIN, A., HAN, R., and GRUNWALD, D., “Privacy-aware location sensor networks,” in *hotos9*, (Lihue, Hawaii), May 2003.
- [80] GU, W., EISENHAEUER, G., KRAEMER, E., SCHWAN, K., STASKO, J., and VETTER, J., “Falcon: On-line monitoring and steering of large-scale parallel programs,” *Frontiers*, February 1995.
- [81] HAUPT, T., AKARSU, E., and FOX, G., “Webflow: A framework for web based meta-computing,” in *High-Performance Computing and Networking, 7th International Conference (HPCN Europe)*, (Amsterdam, The Netherlands), pp. 291–299, April 1999. Also published in *Lecture Notes in Computer Science*, Vol. 1593, Springer, 1999.
- [82] HENGARTNER, U. and STEENKISTE, P., “Access control to information in pervasive computing environments,” in *hotos9*, (Lihue, Hawaii), May 2003.
- [83] HEWLETT-PACKARD, “Adaptive Enterprise.” <http://h71028.www7.hp.com/enterprise/cache/6842-0-0-0-121.html>.
- [84] HEWLETT-PACKARD DEVELOPMENT COMPANY, L.P., “HP OpenView Self-Healing Services Whitepaper.” http://www.managementsoftware.hp.com/services/selfhealing_whitepaper.%pdf.
- [85] HONG, J. I. and LANDAY, J. A., “An architecture for privacy-sensitive ubiquitous computing,” in *MobiSys '04: Proceedings of the 2nd international conference on Mobile systems, applications, and services*, (New York, NY, USA), pp. 177–189, ACM Press, 2004.
- [86] HOUSLEY, R., FORD, W., POLK, W., and SOLO, D., “Internet x.509 public key infrastructure - certificate and crl profile,” RFC 2459, Network Working Group, January 1999.
- [87] HOWARD, M. and LEBLANC, D. C., *Writing Secure Code*. Microsoft Press, second ed., 2002.
- [88] HUANG, Y. and LEE, W., “A cooperative intrusion detection system for ad hoc networks,” in *Proceedings of the ACM Workshop on Security of Ad Hoc and Sensor Networks (SASN 2003)*, (Fairfax, VA), ACM, October 2003.
- [89] HUEBSCHER, M. C. and MCCANN, J. A., “Adaptive middleware for context-aware applications in smart-homes,” in *Proceedings of the 2nd workshop on Middleware for pervasive and ad-hoc computing*, pp. 111–116, ACM Press, 2004.
- [90] IBM CORPORATION, “alphaWorks: Policy Management for Autonomic Computing.” <http://alphaworks.ibm.com/tech/pmac/>.
- [91] IBM CORPORATION, “IBM Autonomic Computing.” <http://www.ibm.com/autonomic/>.

- [92] IBM CORPORATION, “IBM Autonomic computing products and services: IBM WebSphere software.” <http://www.ibm.com/autonomic/websphere.shtml>.
- [93] IBM CORPORATION, “Websphere.” <http://www.ibm.com/websphere/>.
- [94] ISERT, C. and SCHWAN, K., “ACDS: Adapting computational data streams for high performance,” in *Proceedings of International Parallel and Distributed Processing Symposium (IPDPS)*, May 2000.
- [95] JACKSON, B. S. A. W., STRAYER, W. T., ZHOU, W., ROCKWELL, R. D., and PARTRIDGE, C., “Smart packets: applying active networks to network management,” *ACM Transactions on Computer Systems*, vol. 18, pp. 67–88, February 2000.
- [96] JAEGER, T., PRAKASH, A., LIEDTKE, J., and ISLAM, N., “Flexible control of down-loaded executable content,” *ACM Transactions on Information and System Security*, vol. 2, no. 2, pp. 177–228, 1999.
- [97] JAEGER, T., SAILER, R., and ZHANG, X., “Analyzing integrity protection in the selinux example policy,” in *Proceedings of the 12th USENIX Security Symposium*, (Washington, DC), pp. 59–74, USENIX, August 2003.
- [98] JOSEPH, A. D., DELESPINASSE, A. F., TAUBER, J. A., GIFFORD, D. K., and KAASHOEK, M. F., “Rover: A toolkit for mobile information access,” in *Proceedings of the 15th ACM Symposium on Operating System Principles*, (Copper Mountain, CO), pp. 156–171, ACM Press, December 1995.
- [99] JOSEPHSON, W. K., SIRER, E. G., and SCHNEIDER, F. B., “Peer-to-peer authentication with a distributed single sign-on service,” in *Proceedings of the International Workshop on Peer-to-Peer Systems*, (San Diego, CA), February 2004.
- [100] (JSDL-WG), J. S. D. L. W. G., “Job submission description language.” <https://forge.gridforum.org/projects/jsdl-wg/>, July 2005.
- [101] KAMINSKY, M., SAVVIDES, G., MAZIERES, D., and KAASHOEK, M. F., “Decentralized user authentication in a global file system,” in *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles*, (New York, NY, USA), pp. 60–73, ACM Press, 2003.
- [102] KASIC, C. and WALPOLE, J., “QoS scalability for streamed media delivery,” Tech. Rep. CSE-99-011, Oregon Graduate Institute, September 1999.
- [103] KEETON, K., PATTERSON, D. A., and HELLERSTEIN, J. M., “A case for intelligent disks (idisks),” *SIGMOD Record*, vol. 27, September 1998.
- [104] KEROMYTIS, A. D., PAREKH, J., GROSS, P. N., KAISER, G., MISRA, V., NIEH, J., RUBENSTEIN, D., and STOLFO, S., “A holistic approach to service survivability,” in *SSRS '03: Proceedings of the 2003 ACM workshop on Survivable and self-regenerative systems*, pp. 11–22, ACM Press, 2003.
- [105] KIRIANSKY, V., BRUENING, D., and AMARASINGHE, S., “Secure execution via program shepherding,” in *Proc. 11th USENIX Security Symposium*, (San Francisco, CA), August 2002.

- [106] KONG, J. and SCHWAN, K., “Kstreams: Kernel support for efficient end-to-end data streaming,” Tech. Rep. GIT-CERCS-04-04, Center for Experimental Research in Computer Systems, Georgia Institute of Technology, 2004.
- [107] KRAMER, D., “The Java platform: A white paper,” *Sun Microsystems Inc*, May 1996.
- [108] LEVESON, N. G., *Safeware: System Safety and Computers*. Addison-Wesley, 1995.
- [109] LEVIN, R., COHEN, E., POLLACK, F., CORWIN, W., and WULF, W., “Policy/mechanism separation in hydra,” in *Proceedings of the 5th Symposium on Operating Systems Principles*, November 1975.
- [110] LEVY, H., *Capability-Based Computer Systems*. Digital Press, 1984.
- [111] LI, N., MITCHELL, J., and WINSBOROUGH, W., “Design of a role-based trust-management framework,” in *Proceedings of the 2002 IEEE Symposium on Security and Privacy*, (Oakland, VA), IEEE, May 2002.
- [112] LIU, H. and JACOBSEN, H.-A., “Modeling uncertainties in publish/subscribe systems,” in *Proc. 20th International Conference on Data Engineering*, 2004.
- [113] MANOHAR, D., April 2005. Personal communication.
- [114] MASSACHUSETTS INSTITUTE OF TECHNOLOGY, “MIT project Oxygen.” <http://www.lcs.mit.edu/>.
- [115] MICROSOFT CORPORATION, “Microsoft .net framework.” <http://msdn.microsoft.com/netframework/>.
- [116] MICROSYSTEMS, S., “The Jini[tm] distributed event specification, version 1.0.1,” tech. rep., Sun Microsystems, November 1999. <http://www.sun.com/jini/specs/event101.html>.
- [117] MOCKAPETRIS, P. and DUNLAP, K. J., “Development of the domain name system,” in *Proceedings of the Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM)*, (Stanford, CA), pp. 123–133, ACM, August 1988.
- [118] MURATA, T. and MINSKY, N. H., “On shouting ”Fire!”: Regulating decoupled communication in distributed systems (abstract),” in *Proc. ACM/IFIP/USENIX International Middleware Conference*, (Rio de Janeiro, Brazil), June 2003.
- [119] NEEDHAM, R. and WALKER, R., “The cambridge cap computer and its protection system,” in *Proc. Sixth ACM Symposium on Operating System Principles*, November 1977.
- [120] NETEGRITY, “S2ML: The xml standard for describing and sharing security services on the internet,” tech. rep., 2001.
- [121] NOBLE, B. D., SATYANARAYANAN, M., NARAYANAN, D., TILTON, J. E., FLINN, J., and WALKER, K. R., “Agile application-aware adaptation for mobility,” in *Proceedings of the 16th ACM Symposium on Operating System Principles*, (Saint Malo, France), pp. 276–287, October 1997.

- [122] OBJECT MANAGEMENT GROUP, “Corba components - volume 1,” Tech. Rep. orbos/99-07-01, OMG, July 1999. <http://www.omg.org/cgi-bin/doc?orbos/99-07-01>.
- [123] OLESON, V., EISENHAUER, G., PU, C., SCHWAN, K., PLALE, B., and AMIN, D., “Operational information systems - an example from the airline industry,” in *First Workshop on Industrial Experiences with System Software*, (San Diego, CA), pp. 1–10, October 2000.
- [124] OLESON, V., SCHWAN, K., EISENHAUER, G., PLALE, B., PU, C., and AMIN, D., “Operational information systems - an example from the airline industry,” in *Proceedings of the First Workshop on Industrial Experiences with Systems Software (WIESS'2000)*, (San Diego, CA), USENIX Society, October 2000.
- [125] OUSTERHOUT, J. K., *Tcl and the Tk Toolkit*. Addison-Wesley, 1994.
- [126] PERRY, D. E. and WOLF, A. L., “Foundations for the study of software architecture,” *ACM SIGSOFT Software Engineering Notes*, vol. 17, October 1992.
- [127] PETERSON, D. S., BISHOP, M., and PANDEY, R., “A flexible containment mechanism for executing untrusted code,” in *Proc. 11th USENIX Security Symposium*, (San Francisco, CA), August 2002.
- [128] PLALE, B., EISENHAUER, G., SCHWAN, K., HEINER, J., MARTIN, V., and VETTER, J., “From interactive applications to distributed laboratories,” *IEEE Concurrency*, vol. 6, no. 2, 1998.
- [129] POLETTI, M., ENGLER, D., and KAASHOEK, M. F., “tcc: A template-based compiler for ‘c,’” in *Proceedings of the First Workshop on Compiler Support for Systems Software (WCSS)*, February 1996.
- [130] PROVOS, M., “Improving host security with system call policies,” in *Proc. 12th USENIX Security Symposium*, (Washington, DC), August 2003.
- [131] PU, C., AUTREY, T., BLACK, A., CONSEL, C., COWAN, C., INOUE, J., KETHANA, L., WALPOLE, J., and ZHANG, K., “Optimistic incremental specialization: Streamlining a commercial operating system,” in *Proceedings of the Fifteenth Symposium on Operating Systems Principles*, (Colorado), ACM, December 1995.
- [132] RADICATI, S., “X.500 directory services: Technology and deployment,” tech. rep., International Thomson Computer Press, London, UK, 1994.
- [133] RIEDEL, E. and GIBSON, G., “Active disks - remote execution for network-attached storage,” Tech. Rep. CMU-CS-97-198, School of Computer Science, Carnegie Mellon University, Pittsburgh, Pennsylvania, December 1997.
- [134] RITCHIE, D. M., “The evolution of the Unix time-sharing system,” *AT&T Bell Laboratories Technical Journal*, vol. 63, pp. 1577–1593, October 1984.
- [135] RIVEST, R., “The md5 message-digest algorithm (rfc 1321),” April 1992.

- [136] RODRIGUES, C., LOYALL, J., and SCHANTZ, R., “Quality objects (QuO): Adaptive management and control middleware for end-to-end QoS,” in *OMG’s First Workshop on Real-Time and Embedded Distributed Object Computing*, (Falls Church, Virginia), July 2000.
- [137] ROSU, D. I., SCHWAN, K., YALAMANCHILI, S., and JHA, R., “On adaptive resource allocation for complex real-time applications,” in *18th IEEE Real-Time Systems Symposium, San Francisco, CA*, pp. 320–329, IEEE, Dec. 1997.
- [138] ROSU, M.-C., SCHWAN, K., and FUJIMOTO, R., “Supporting parallel applications on clusters of workstations: The virtual communication machine-based architecture,” *Cluster Computing, Special Issue on High Performance Distributed Computing*, vol. 1, pp. 51–67, January 1998.
- [139] SAWANT, N., SCHARVER, C., LEIGH, J., JOHNSON, A., REINHART, G., CREEL, E., BATCHU, S., BAILEY, S., and GROSSMAN, R., “The tele-immersive data explorer: A distributed architecture for collaborative interactive visualization of large datasets,” in *Proceedings of the Fourth International Immersive Projection Technology Workshop*, (Ames, Iowa), 2000.
- [140] SCHLUMBERGER LIMITED <http://www.schlumberger.com>.
- [141] SCHNEIER, B., *Secrets and Lies: Digital Security in a Networked World*. John Wiley & Sons, 2000.
- [142] SCHNEIER, B. and KELSEY, J., “Secure audit logs to support computer forensics,” *ACM Transactions on Information and System Security*, vol. 2, pp. 159–176, May 1999.
- [143] SEGALL, B. and ARNOLD, D., “Elvin has left the building: A publish/subscribe notification service with quenching,” in *Proceedings of the AUUG (Australian users group for Unix and Open Systems) 1997 Conference*, September 1997.
- [144] SESHASAYEE, B., SCHWAN, K., and WIDENER, P., “SOAP-binQ: High-Performance SOAP with Continuous Quality Managementn,” in *Proc. 24th International Conference on Distributed Computing Systems (ICDCS)*, (Tokyo, Japan), March 2004.
- [145] SHAPIRO, J. S., SMITH, J. M., and FARBER, D. J., “EROS: a fast capability system,” in *Proc. 17th ACM Symposium on Operating Systems Principles*, (Kiawah Island, South Carolina), December 1999.
- [146] SHAW, M. and GARLAN, D., *Software Architecture: Perspectives on an Emerging Discipline*. Prentice-Hall, 1996.
- [147] SHEN, Y., LAM, T., LIU, J.-C., and ZHAO, W., “On the confidential auditing of distributed computing systems,” in *Proc. 24th International Conference on Distributed Computing Systems*, 2004.
- [148] SMITH, M., GOOD, G., WELTMAN, R., and HOWES, T., “Persistent search: a simple LDAP change notification mechanism,” working document, IETF, November 2000. draft-smith-psearch-ldap-00.txt.

- [149] SMITH, W., WAHEED, A., MEYERS, D., and YAN, J., “An evaluation of alternative designs for a grid information service,” in *Proceedings of the 9th International Symposium on High Performance Distributed Computing (HPDC-9)*, (Pittsburgh, PA), IEEE, August 2000.
- [150] STEEN, M. V., HAUCK, F. J., and TANENBAUM, A. S., “Locating objects in wide-area systems,” *IEEE Communication Magazine*, pp. 104–109, January 1998.
- [151] STROM, R., BANAVAR, G., CHANDRA, T., KAPLAN, M., MILLER, K., MUKHERJEE, B., STURMAN, D., and WARD, M., “Gryphon: An information flow based approach to message brokering,” in *International Symposium on Software Reliability Engineering '98 Fast Abstract*, 1998.
- [152] SUMMERS, R. C., *Secure Computing: threats and safeguards*. McGraw-Hill, 1997.
- [153] SUN MICROSYSTEMS INC., “Enterprise Java Beans technology.” <http://java.sun.com/products/ejb/>.
- [154] TENNENHOUSE, D. L., SMITH, J. M., SINCOSKIE, W. D., WETHERALL, D. J., and MINDEN, G. J., “A survey of active network research,” *IEEE Communications Magazine*, vol. 35, pp. 80–86, January 1997.
- [155] TESAURO, G., CHESS, D. M., WALSH, W. E., DAS, R., SEGAL, A., WHALLEY, I., KEPHART, J. O., and WHITE, S. R., “A multi-agent systems approach to autonomic computing,” in *AAMAS '04: Proceedings of the Third International Joint Conference on Autonomous Agents and Multiagent Systems*, pp. 464–471, IEEE Computer Society, 2004.
- [156] TRIPATHI, A. R., AHMED, T., KUMAR, R., and JAMAN, S., “Design of a policy-driven middleware for secure distributed collaboration,” in *Proc. 22nd International Conference on Distributed Computing Systems*, 2002.
- [157] TRUSTED COMPUTING PLATFORM ALLIANCE, “TCPA main specification v1.1b.” <http://www.trustedcomputing.org/>.
- [158] VAHDAT, A., DAHLIN, M., ANDERSON, T., and AGGARWAL, A., “Active names: flexible location and transport of wide-area resources,” in *Proceedings of USENIX Symp. on Internet Technology & Systems*, October 1999.
- [159] VAN ENGELEN, R. and GALLIVAN, K., “The gSOAP toolkit for web services and peer-to-peer computing networks,” in *Proc. 2nd IEEE International Symposium on Cluster Computing and the Grid*, 2002.
- [160] VAN STEEN, M., HOMBURG, P., , and TANENBAUM, A., “Globe: A wide-area distributed system,” *IEEE Concurrency*, vol. 7, pp. 70–78, January-March 1999.
- [161] VIXIE, P., “A mechanism for prompt notification of zone changes (dns notify),” RFC 1996, Network Working Group, August 1996.
- [162] VON EICKEN, T., CULLER, D. E., GOLSDTEIN, S. C., and SCHAUSER, K. E., “Active messages: a mechanism for integrated communication and computation,” Report UCB/CSD 92/675, Univeristy of California, Berkley, March 1992.

- [163] WAHBE, R., LUCCO, S., ANDERSON, T. E., and GRAHAM, S. L., “Efficient software-based fault isolation,” *ACM SIGOPS Operating Systems Review*, vol. 27, pp. 203–216, December 1993.
- [164] WALLACH, D. S., BALFANZ, D., DEAN, D., and FELTEN, E. W., “Extensible security architectures for Java,” in *Proceedings of the 16th ACM Symposium on Operating System Principles*, (Saint-Malo, France), pp. 116–128, December 1997.
- [165] WANG, L., JAJODIA, S., and WIJESSEKERA, D., “Securing OLAP data cubes against privacy breaches,” in *Proc. IEEE Symposium on Security and Privacy*, 2004.
- [166] WELCH, V., SIEBENLIST, F., FOSTER, I., BRESNAHAN, J., CZAJKOWSKI, K., GAWOR, J., KESSELMAN, C., MEDER, S., PEARLMAN, L., and TUECKE, S., “Security for grid services,” in *Proceedings of the Twelfth International Symposium on High Performance Distributed Computing (HPDC-12)*, IEEE, IEEE Press, June 2003.
- [167] WELSH, M., BASU, A., and EICKEN, T. V., “Incorporating memory management into user-level network interfaces,” in *Proceedings of Hot Interconnects V*, pp. 27–36, 1997.
- [168] WEST, R., SCHWAN, K., TACIC, I., and AHAMAD, M., “Exploiting temporal and spatial constraints on distributed shared objects,” in *Proceedings of the IEEE International Conference on Distributed Computing Systems*, (Baltimore, Maryland), IEEE, May 1997.
- [169] WETHERALL, D. J., GUTTAG, J. V., and TENNENHOUSE, D. L., “ANTS: a toolkit for building and dynamically deploying network protocols,” in *The First IEEE Conference on Open Architectures and Network Programming (OpenArch’1998)*, (San Francisco, CA), April 1998.
- [170] WIDENER, P., EISENHAEUER, G., SCHWAN, K., and BUSTAMANTE, F. E., “Open metadata formats: Efficient xml-based communication for high-performance computing,” *Cluster Computing: The Journal of Networks, Software Tools, and Applications*, no. 5, pp. 315–324, 2002.
- [171] WIDENER, P., SCHWAN, K., and BUSTAMANTE, F., “Differential data protection in dynamic distributed applications,” in *Proceedings of the 2003 Annual Computer Security Applications Conference*, (Las Vegas, NV), December 2003.
- [172] WIDENER, P., SCHWAN, K., and EISENHAEUER, G., “Open metadata formats for fast communication,” Tech. Rep. GIT-CC-00-21, College of Computing, Georgia Institute of Technology, Atlanta, Georgia, 2000.
- [173] WIDENER, P. M., BRIERCHECK, G. S., HIMMICH, S. C., MCKEE, S. A., PERI, R. V., and WULF, W. A., “The WM protection mechanism,” Tech. Rep. TR-92-28, University of Virginia Department of Computer Science, Charlottesville, Virginia, August 1992.
- [174] WOLF, M., CAI, Z., HUANG, W., and SCHWAN, K., “Smart pointers: Personalized scientific data portals in your hand,” in *Proceedings of Supercomputing 2002*, November 2002.

- [175] YARVIS, M., REIHER, P., and POPEK, G. J., “Conductor: A framework for distributed adaptation,” in *Proceedings of the 7th Workshop on Hot Topics in Operating Systems (HotOS-VII)*, (Rio Rico, AZ), March 1999.
- [176] YEONG, W., “Lightweight directory access protocol,” RFC 1777, Network Working Group, March 1995.
- [177] YNGVE, V., “The chicago magic number computer,” tech. rep., University of Chicago Institute for Computer Research, November 1968.
- [178] YU, H., ESTRIN, D., and GOVINDAN, R., “A hierarchical proxy architecture for internet-scale event services,” in *Proceedings of WETICE’99*, (Stanford, CA), June 1999.
- [179] ZENEL, B. and DUCHAMP, D., “A general purpose proxy filtering mechanism applied to the mobile environment,” in *Proceedings of Mobile Computing (MOBICOM) 97*, (Budapest, Hungary), pp. 248–259, September 1997.
- [180] ZHANG, G. and PARASHAR, M., “Dynamic context-aware access control for grid applications,” in *Proc. Fourth International Workshop on Grid Computing (GRID’03)*, (Phoenix, Arizona), November 2003.
- [181] ZHOU, D. and SCHWAN, K., “JECho - Supporting Distributed High Performance Applications with Java Event Channels,” in *Proc. International Parallel and Distributed Processing Symposium*, May 2001.
- [182] ZHOU, D., SCHWAN, K., EISENHAEUER, G., and CHEN, Y., “JECho - interactive high performance computing with Java event channels,” in *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS 2001)*, April 2001.

VITA

Patrick McCall Widener was born in Johnson City, Tennessee, USA on September 22, 1968. He received the Bachelor of Science degree from James Madison University in 1990 and the Master of Computer Science degree from the University of Virginia in 1992.

After graduation from the University of Virginia, he worked in private industry as a software developer and consultant for several years. During this time he participated in many different aspects of the software development process, including team leadership, new technology pioneering, and business process reengineering. Working mostly in the telecommunications industry, he was responsible for the development of innovative and revolutionary tools for use by both colleagues and customers. While leading the Software Development and Integration team in Bell Atlantic's Internet Center of Excellence, he oversaw the architecture and construction of the prototype of Bell Atlantic's first Internet customer bill presentation system.

In July 2005, under the supervision of Professor Karsten Schwan, he completed his Ph.D. dissertation entitled "Dynamic Differential Data Protection for High-Performance and Pervasive Applications".