Dynamic Differential Data Protection for High Performance and Pervasive Applications

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

Abstract

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 implication of these requirements is that static policy and mechanism definitions are unsuitable for the design of modern software systems. This paper addresses the protection mechanisms of such systems, describing a novel approach to enabling the protection of key applications components and sensitive data in distributed applications. The approach, termed Dynamic Differential Data Protection (D3P), deploys context-sensitive, application-specific protection functionality at runtime to enforce restrictions in data access and manipulation. D3P is suitable for use in zero/low-downtime environments, appropriate for high-performance computing tasks and highly-scalable architectural patterns (such as publish/subscribe), and is deployable across a wide variety of OS and machine platforms. We introduce the need for D3P, using sample applications from the HPC and pervasive computing domains to illustrate the solutions it makes possible, and describe how D3P has been integrated into modern middleware. We demonstrate D3P's ability to capture individual end-users' or components' needs for data protection. Finally, we present experimental evaluations which quantify the performance implications of using D3P in data-intensive applications.

1. Introduction

Modern distributed applications serve an increasingly connected world, providing services that adapt to changing execution environments and end user needs. Openness, connectedness, and dynamic change create new challenges for the protection of application components and sensitive data:

- For businesses, how to share data with partners or subcontractors, without divulging proprietary or company-critical information? An example from the airline industry is the need to share selected passenger data with catering subcontractors, without compromising passengers' privacy concerns [28]. In oil exploration, reservoir simulation data produced in computer centers must be shared with remote, on-site drilling teams [30].
- For scientists, how to exchange data relevant to current collaborations and/or coordinate access to shared research equipment [9, 31], without divulging high-resolution data critical to individual scientific processes (e.g., the publication of new results or insights)?
- In sensor applications, how to distribute images captured by remote sensors [18] so as to meet the needs of the diverse applications using this data, ranging from simple monitoring or surveillance, to detailed tasks like tracking (e.g., radar images) or face recognition?

The requirements imposed by dynamic, distributed applications have given rise to new technical approaches and solutions, ranging from efficient group authentication in grid computing [33], to distributed trust models and mechanisms in peer-to-peer systems [25, 4, 13], to new solutions for access auditing [19]. In comparison to such research, this paper focuses on issues in 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. Limitations in data security resulting from this approach are discussed in Section 4.

The fundamental concept underlying (1)-(4) is that of *dynamic, differential data protection* (D3P). D3P derives its basic data access model from prior work on object- and capability-based models of data protection [24], where typed data is accessed and manipulated only by well-defined operations. To accommodate modern distributed applications, however, D3P permits the code fragments that operate on data objects to be generated and deployed dynamically, thereby allowing applications to react *dynamically* to changing end user needs or execution conditions, constrained only by applications' security policies. Furthermore, D3P provides *differential* support for protecting application data and/or components. That is, based on metadata about the application-level information being accessed, the granularity of access control provided by D3P can be adjusted as needed by applications, again bounded only by security policies.

D3P's current middleware-level implementation uses cryptographic techniques to protect capabilities, but it does not encrypt the data being manipulated. As a result, D3P does not address data integrity or privacy, which poses a problem when network links are at risk for eavesdropping and/or "sniffing". On a single machine, such properties could be provided by placing D3P middleware into a protection ring 'below' the application and by using TPM-based data encryption, perhaps using platforms that implement the requirements stated by the trusted platform architecture [21]. Alternatively, stronger guarantees for D3P could be implemented by dynamically extending operating system kernels and requiring applications to make system calls to access their data, representing the application-level codes operating on data objects with safe code execution facilities like software fault isolation [32] or the kernel plugins described in [15], requiring applications to make system calls to access their data, and using kernel-to-kernel communications to transport data across different machines. Furthermore, standard techniques for executing user-supplied code are concerned with protecting the code, the system executing the code, or both. In contrast, D3P provides data rather than code protection. Access to the data items manipulated by the user-supplied code is controlled, but the safety of that code is not guaranteed. If needed, standard mechanisms like sandboxing can be incorporated into the D3P model and its implementation. Innovative approaches that apply hardware-based segmentation to user-supplied code [16] are also compatible with D3P. Other standard tools leveraged by D3P include X.509 certificate technology for identification/authentication of principals. In this fashion, D3P is compatible with existing large-scale security architectures such as GSI[8]. Other authentication technologies such as Kerberos are not directly embraced by D3P, although there is in principle nothing preventing the extension of the system in such directions.

The main contributions of this paper are its development of the D3P concept and the application of the concept to both high performance and pervasive distributed applications [35, 34, 6] (see Section 2), demonstrating its straightforward yet flexible approaches to data protection in representative usage scenarios. Section 3 provides an abstract definition of the protection and extension model defined by D3P. Interesting implementation details are reported in Section 4. Experimental results, presented in Section 5, demonstrate that applications gaining advantages from the new functionality provided by D3P do so with only small losses in communication performance. This is due to the fact that D3P's implementation does not significantly affect the "fast path" of data transfer, which means that D3P can be used both with high performance, data-rich scientific or engineering applications and with highly resource-constrained pervasive systems.

2. A data protection perspective on applications

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

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, extension, and specialization [10, 29].

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 applications transport and manipulate.

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 [23] allow for an arbitrary or unknown set of users. Reflective/introspective development frameworks [27, 20, 26] and open standards for control transfer [3] and data description and exchange [1] 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.

The primary contribution of D3P is that it allows us to 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 options to deal with changing user populations (to either allow unknown users or not), data definitions (preventing the use of low-display-capability handheld 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 only those fields, instead of their being restricted from the type altogether. D3P provides a method of avoiding such all-or-nothing decisions.

The following sections examine the need for and utility of D3P in two representative applications: (1) a surveillance system that uses remote cameras to capture and inspect images and (2) a high performance application used by scientists to collaborate in real-time via meaningful remote data visualizations.

2.1. Pervasive applications: surveillance

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 [7] 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 analyses, 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) [2].

A basic tradeoff 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 tradeoffs. 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 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 tradeoff 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 1. 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.

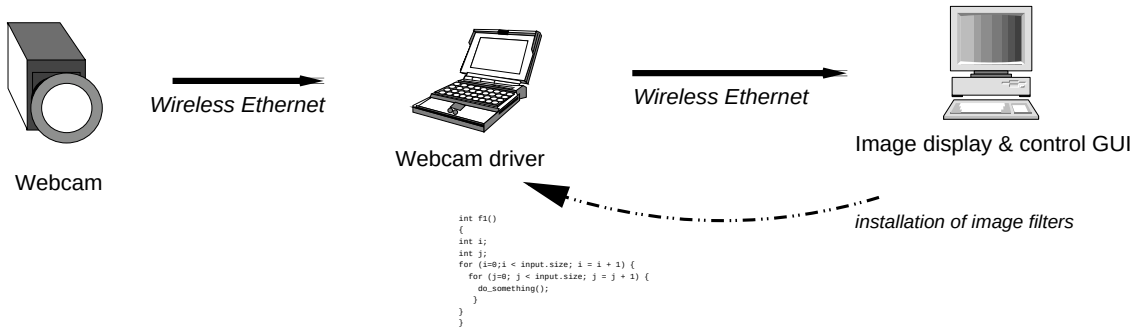

Figure 2. The AVS (Active Video Streams) application.

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 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. A typical scenario 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. Note that D3P does not rely on specific operating system support (as might be done with a set of protected types in SELinux [22], for example). Standard identification methods (X.509 certification) are used, decoupling applications such as AVS from a specific operating system's access control matrix. At the same time, D3P's flexible support for incorporating system- and application-specific information into the capability generation process allows the AVS developer to use established policy databases.

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. Application in which such data exchanges take place are those built with the Smart-Pointer framework for real-time scientific collaboration developed by our group [35]. Here, 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.

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 control 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.
- The aim of D3P is to provide feature-complete middleware facilities for implementing applications of the scale and scope of SmartPointer. Standard cryptographic techniques provide a common security "vocabulary". Modular implementation of policy management allows flexibility in addressing requirements from diverse user communities whose needs and constraints may not have much in common.

We have described how the D3P approach can benefit designers, developers and users in both the high-performance and pervasive application domains. In the next section, we present the fundamental concepts of D3P that make these benefits possible.

3. The D3P model

D3P is a model for a *protection mechanism*. By separating policy from mechanism, it preserves the "library"-like nature of middleware implementing D3P principles. This section relates the D3P concepts, the end goal being to build middleware with properties that match the needs of future applications while also providing suitable support for data access protection.

3.1. 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 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.

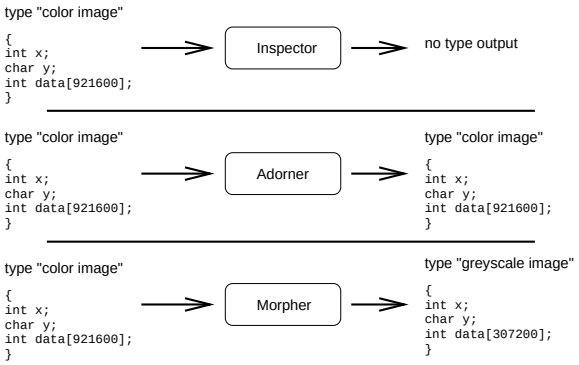

Figure 3. Inspectors, Adorners, and Morphers take different actions on input and output types.

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 chain*. Handler chains, as with single handlers, have a type which is expressed as a tuple. A handler chain type tuple consists of the input type of the first handler in the chain and the output type (or `nil`) of the last handler in the chain. The boolean result value of the chain is the value provided by the last handler in the chain.

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 chains, their default organization is a logical AND across the chain. In this mode, the boolean return value from each handler in the chain must be `true` in order for the remaining handlers in the chain to execute. If each handler returns `true`, the data is forwarded across the link. For purposes of determining whether or not data is forwarded, the chain is treated as a "black box". The boolean expression represented by the "black box" of the handler chain can be modified by inserting different boolean and grouping operators after the chain is constructed.

Complex functionality can be encoded and manipulated using chains of handlers. For example, any symmetric data manipulation task (such as encryption/decryption) can be modeled in D3P as the installation of matched pairs of handler chains at appropriate link endpoints.

3.2. Referring to objects

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 Authority. Authority 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 Authority; 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 chain provided by D3P allows chains to be manipulated and reasoned about in the same manner as single handlers. A handler (or chain) 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

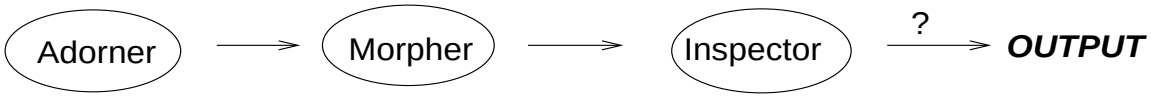

Figure 4. Handlers can be combined in *chains* to achieve a series of effects.

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

3.3. 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. Finally, 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.

4. Realizing D3P in publish/subscribe middleware

The previous section described the abstract middleware model of D3P. In order to evaluate these ideas, we have produced a concrete D3P implementation in 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 implement D3P for pub/sub 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. We next briefly describe its fundamentals.

4.1. ECho fundamentals

ECho [14] 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[17], 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 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) [5]. PBIO allows high-level description of data

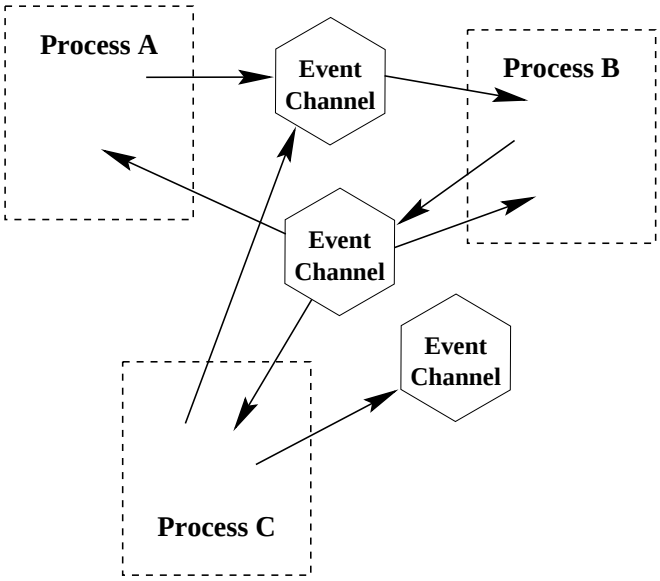

Figure 5. Processes using Event Channels for communication.

to be efficiently represented and transmitted in binary form. PBIO transparently handles binary translation issues such as differing machine word sizes or word endian-ness, provides facilities for compile-time or run-time type definition, and performs type reflection and conversion between compatible types.

4.2. ECho implementation

ECho provides a set of abstractions to the programmer and an API that can be used to create, dispose of, and manipulate them. We implement D3P underneath those abstractions and provide whatever additional interfaces might be required. This choice is made for two reasons. First, there are several mature applications already supported by ECho; by maintaining a compatible interface, we gain access to this body of code. Second, experiences in the high performance domain demonstrate the propensity of application developers to discard any extraneous software layers like those implementing security in favor of improved performance. By operating 'underneath' the existing ECho interface, D3P protection features are incorporated by default. By carefully integrating D3P 'into' the implementation of ECho, undue performance penalties (see Section 5) are avoided.

The primary task necessary to implement D3P in ECho was to provide capabilities for ECho objects. This is necessary in order to enforce protection restrictions on those objects. The canonical definition of a capability is a reference to an object combined with a set of rights relative to that object. To make an ECho object into a capability, two things are necessary: secure the object against forgery, and add rights information. Cryptographic techniques are used to secure each ECho object of interest. Specifically, each object is signed by a Security Manager (a trusted module) to indicate its validity. This signature is verified by the middleware each time the object is used; malicious users are thereby prevented from forging protected ECho objects. The rights field for each object contains both universal and type-specific rights. For this purpose, ECho capabilities also include an object descriptor that identifies their type.

Note that this will not deter an attacker that is able to scan all memory at will, as might happen if host administration security is compromised. Such protection is beyond the scope or ability of any middleware system. However, it does prevent local forgery and copying of ECho objects. Language-level opacity is used to prevent direct programmatic manipulation of ECho objects. Finally, current research efforts[12, 11] into language-level security provide additional options for increased safety.

The "bootstrap" ECho object is the `EControlContext` (ECC); all other ECho operations require its cooperation. The API for creating an ECC was modified to send a list of attributes to the Security Manager, which returns a *protected* ECC object. 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 capability provided for the ECC 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. If so, a new protected `EChannel` is created and returned to the caller.

Capabilities for PBIO types are also obtained via the Security Manager. PBIO defines a local dictionary of types known as an `IOContext`; this object is cryptographically secured in the same way as other capabilities.

Event submission in unprotected ECho requires an `EControlContext`and a specification of the type of the event. The analogous call in D3P-enabled ECho is to provide capabilities for both the `EControlContext`and the `IOContext`containing the type in question. This call is in the data transfer path; however, the only significant additional computation required is the signature verification for both capabilities. Revocation of capabilities is currently not handled, but could be implemented by embedding a revocation list in the `EControlContext`object (to be updated periodically or on demand).

ECho directly supports the D3P concept of handlers, through what are called *derived* channels. A derived channel has had a handler installed on it; the handler executes

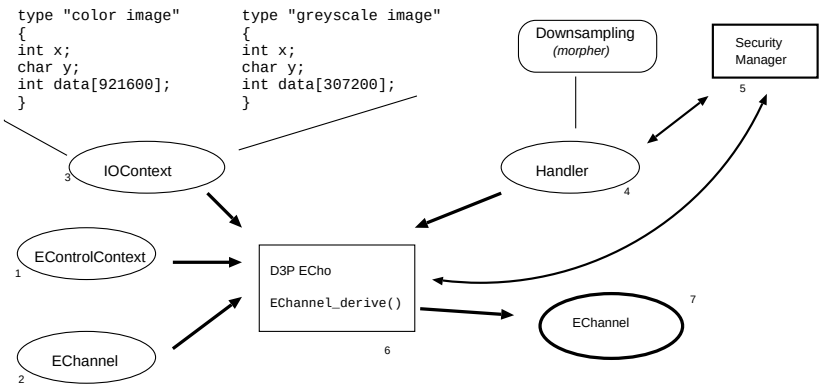

Figure 6. Installing a handler in AVS requires the use of several D3P objects. An *EControlContext* (1) is needed in order to create any other protected ECho objects. Capabilities for the original, unmodified *EChannel* (2), the PBIO types involved (the type used in the original channel and the one to 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 the Security Manager (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.

once for each event passing through the channel. 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 the Security Manager; 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 the Security Manager 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.

4.3. Bootstrapping capabilities in ECho

Capability systems typically assume the presence of a TCB or trusted module that has the power to create, modify, and dispose of them. As the creation or destruction of a capability is a direct representation of an access control decision, the presence of a security policy is also implied. We have established a Security Manager (SM) (an instance of Authority from the D3P model) to encapsulate capability actions and any policy manipulation.

We did this for three reasons. First, we wish to remain open to different authentication/authorization schemes, and abstracting away these details to a remotely located entity avoids dependencies. Note that this does not necessarily imply a network round-trip, unlike our previous designs[34]. Our current implementation performs authentication and authorization against `/etc/passwd`, and integrating with other authentication systems such as Kerberos or Grid security infrastructures like GSI[8] is as simple as providing an appropriate API. Secondly, it is best to keep trusted code to a minimum, and to keep the size of trusted code modules as small as possible. Since any SM-like process would obviously need to be trusted, it makes sense to separate it from the rest of the middleware. Finally, we wish to provide implementation tools and mechanisms, as opposed to tools for defining, manipulating, and integrating security policies. We view a modularized SM as a way to ensure that policy questions remain "somebody else's problem".

5. Experimental evaluation

We first characterize the overhead of our D3P implementation in ECho by measuring time necessary 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 some 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 our mechanism is that its performance overheads are not in the critical path of data transfer. Once access to the channel has been established

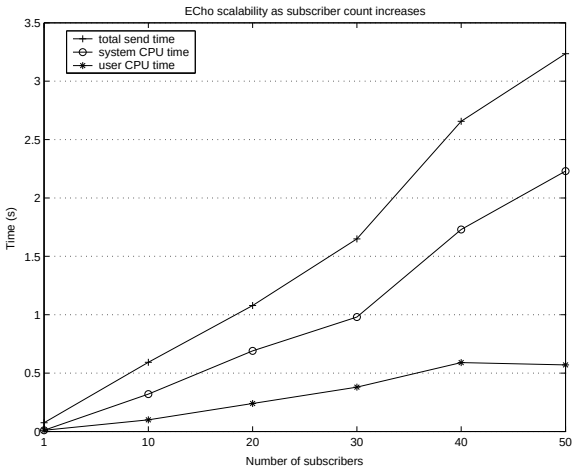

Figure 7. ECho scaleability with no D3P features.

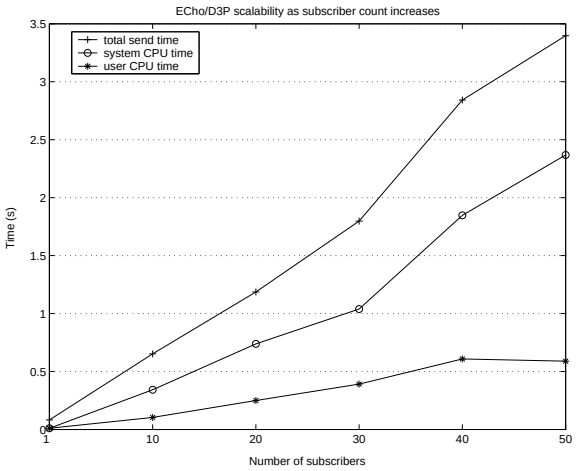

Figure 8. ECho scalability where increasing numbers of clients use the D3P interfaces.

or a handler installed, data transfer proceeds at speeds limited only by the underlying middleware or network. Even in those situations where the mechanism does have a performance impact, that impact is minimal.

5.1. ECho scalability

In this series of experiments, we show that the addition of D3P functionality does not impact the scalability of the ECho middleware. Previous research [6] has demonstrated the scalability of the unmodified ECho middleware. We repeated our previous experiment using the D3P version of the ECho middleware. Results appear in Figure 8. As expected, absolute performance is worse than the unprotected ECho case. 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 the network stack).

Figure 7 shows that ECho performance scales roughly linearly as the number of clients per channel increases.

5.2. Application performance

This experiment shows that performance of a typical pervasive application does not suffer when using D3P. We measure 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 9 shows that, as the 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.

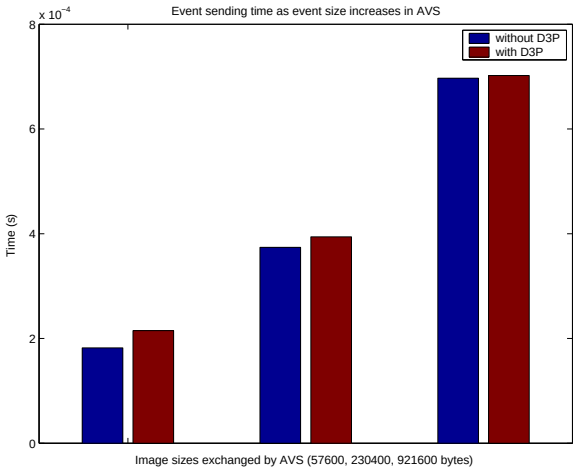

Figure 9. ECho/D3P scalability as event sizes increase.

6. Conclusion

Our primary contribution in this paper has been the presentation of the D3P approach to providing data protection in high performance and pervasive applications. We have described the underlying conceptual model of D3P and examined how it can address application requirements for sample applications in both the high performance and pervasive domains. We have described an implementation of the D3P approach in a mature publish/subscribe middleware package. We have demonstrated that the data protection benefits of D3P can be made available to developers without serious performance penalties.

In the future, we intend to enhance both the D3P model and its reference implementation. We are particularly interested in exploring ways to use D3P with innovative software isolation/virtualization techniques. Integration with security policy research and policy description languages also remains a topic for future attention.

Acknowledgments

Greg Eisenhauer provided valuable insight into the design and architecture of the ECho middleware system.

This work was supported in part by an Intel Foundation Graduate Fellowship Award.

References

[1] The extensible markup language (XML). http://www.w3.org/TR/1998/REC-xml-19980210.

[2] The infopipe toolkit. http://www.cc.gatech.edu/projects/infosphere/software/.

[3] Simple Object Access Protocol. http://www.w3.org/TR/SOAP/.

[4] M. Blaze, J. Feigenbaum, J. Ioannidis, and A. Keromytis. 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, pages 185–210. Springer-Verlag, Berlin, 1999.

[5] F. Bustamante, G. Eisenhauer, K. Schwan, and P. Widener. Efficient wire formats for high performance computing. In *Proceedings of Supercomputing 2000*, November 2000.

[6] F. Bustamante, P. Widener, and K. Schwan. Scalable directory services using proactivity. In *Proceedings of Supercomputing 2002*, Baltimore, MD, November 2002.

[7] F. E. Bustamante. *The Active Streams Approach To Adaptive Distributed Applications and Services*. PhD thesis, Georgia Institute of Technology, November 2001.

[8] R. Butler, D. Engert, I. Foster, C. Kesselman, S. Tuecke, J. Volmer, and V. Welch. A national-scale authentication infrastructure. *IEEE Computer*, 33(12):60–66, 2000.

[9] U. Catalyurek, M. Benyon, C. Chang, T. Kurc, A. Sussman, and J. Saltz. The virtual microscope. *IEEE Transactions on Information Technology in Biomedicine*, 7(4):230–248, 2003.

[10] C. Chambers, S. J. Eggers, J. Auslander, M. Philipose, M. Mock, and P. Pardyak. Automatic dynamic compilation support for event dispatching in extensible systems. In *Proceedings of the Workshop on Compiler Support for Systems Software (WCSSS'96)*. ACM, February 1996.

[11] M. Christodorescu and S. Jha. Static analysis of executables to detect malicious patterns. In *Proceedings of the 12th USENIX Security Symposium*, pages 169–186, Washington, DC, August 2003. USENIX.

[12] C. Cowan, S. Beattie, J. Johansen, and P. Wagle. Pointguard: Protecting pointers from buffer overflow vulnerabilities. In *Proceedings of the 12th USENIX Security Symposium*, pages 91–104, Washington, DC, August 2003. USENIX.

[13] D.Boneh, X. Ding, G. Tsudik, and B. Wong. Fast revocation of security capabilities. In *Proceedings of the 2001 USENIX Security Symposium*. USENIX, 2001.

[14] G. Eisenhauer, F. E. Bustamante, and K. Schwan. Event services in high performance systems. *Cluster Computing: The Journal of Networks, Software Tools, and Applications*, 4(3):243–252, July 2001.

[15] I. Ganev, G. Eisenhauer, and K. Schwan. Kernel plugins: When a vm is too much. In *Proceedings of the 3rd Virtual Machine Research and Technology Symposium*.

[16] I. Ganev, G. Eisenhauer, and K. Schwan. Kernel plugins: When a vm is too much. In *Proceedings of the Third Virtual Machine Research and Technology Symposium*, May 2004.

[17] O. M. Group. *CORBAservices: Common Object Services Specification*, chapter 4. OMG, 1997. http://www.omg.org.

[18] T. He, B. M. Blum, J. A. Stankovic, and T. Abdelzaher. Aida: Adaptive application-independent data aggregation in wireless sensor networks. *Trans. on Embedded Computing Sys.*, 3(2):426–457, 2004.

[19] Y. Huang and W. Lee. 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, October 2003. ACM.

[20] IBM Corporation. Websphere. http://www.ibm.com/websphere/.

[21] Intel Corporation. Lagrande technology (lt) for safer computing. http://www.intel.com/technology/security/.

[22] T. Jaeger, R. Sailer, and X. Zhang. Analyzing integrity protection in the selinux example policy. In *Proceedings of the 12th USENIX Security Symposium*, pages 59–74, Washington, DC, August 2003. USENIX.

[23] W. K. Josephson, E. G. Sirer, and F. B. Schneider. 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.

[24] R. Levin, E. Cohen, F. Pollack, W. Corwin, and W. Wulf. Policy/mechanism separation in hydra. In *Proceedings of the 5th Symposium on Operating Systems Principles*, November 1975.

[25] N. Li, J. Mitchell, and W. Winsborough. Design of a role-based trust-management framework. In *Proceedings of the 2002 IEEE Symposium on Security and Privacy*, Oakland, VA, May 2002. IEEE.

[26] Microsoft Corporation. Microsoft .net framework. http://msdn.microsoft.com/netframework/.

[27] S. Microsystems. The Jini[tm] distributed event specification, version 1.0.1. Technical report, Sun Microsystems, November 1999. http://www.sun.com/jini/specs/event101.html.

[28] V. Oleson, K. Schwan, G. Eisenhauer, B. Plale, C. Pu, and D. Amin. 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, October 2000. USENIX Society.

[29] C. Pu, T. Autrey, A. Black, C. Consel, C. Cowan, J. Inouye, L. Kethana, J. Walpole, and K. Zhang. Optimistic incremental specialization: Streamlining a commercial operating system. In *Proceedings of the Fifteenth Symposium on Operating Systems Principles*, Colorado, December 1995. ACM.

[30] Schlumberger Limited. http://www.schlumberger.com.

[31] US Department of Energy (SciDAC). The national fusion collaboratory. http://www.fusiongrid.org/.

[32] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham. Efficient software-based fault isolation. *ACM SIGOPS Operating Systems Review*, 27(5):203–216, December 1993.

[33] V. Welch, F. Siebenlist, I. Foster, J. Bresnahan, K. Czajkowski, J. Gawor, C. Kesselman, S. Meder, L. Pearlman, and S. Tuecke. Security for grid services. In *Proceedings of the Twelfth International Symposium on High Performance Distributed Computing (HPDC-12)*. IEEE, IEEE Press, June 2003.

[34] P. Widener, K. Schwan, and F. Bustamante. Differential data protection in dynamic distributed applications. In *Proceedings of the 2003 Annual Computer Security Applications Conference*, Las Vegas, NV, December 2003.

[35] M. Wolf, Z. Cai, W. Huang, and K. Schwan. Smart pointers: Personalized scientific data portals in your hand. In *Proceedings of Supercomputing 2002*, November 2002.