A Middleware Toolkit for Client-Initiated Service Specialization

Greg Eisenhauer, Fabián E. Bustamante, Karsten Schwan *
College of Computing
Georgia Institute of Technology

Abstract

As the Internet matures, streaming data services are taking an increasingly important place alongside traditional HTTP transactions. The need to dynamically adjust the delivery of such services to changes in available network and processing resources has spawned substantial research on application-specific methods for dynamic adaptation, including video and audio streaming applications. Such adaptation techniques are well developed, but they are also highly specialized, with the client (receiver) and server (sender) implementing well-defined protocols that exploit content-specific stream properties. This paper describes our efforts to bring the benefits of such content-aware, application-level service adaptation to all types of streaming data and to do so in a manner that is efficient and flexible. Our contribution in this domain is ECho, a high-performance event-delivery middleware system. ECho's basic functionality provides efficient binary transmission of event data with unique features that support dynamic data-type discovery and service evolution. ECho's contribution to data stream adaptation is in the mechanisms it provides for its clients to customize their data flows through type-safe dynamic server extension.

1 Introduction

Internet-delivered content has evolved significantly in the recent past. Where content once consisted principally of simple text and graphics requested and delivered via HTTP, it has broadened to include streaming content such as Internet telephony, broadcast-style audio and video streams, and stock, sports, and weather updates. Streaming data content tends to be more sensitive to delays in content delivery than HTTP-style traffic, resulting in the need to adapt stream content to provide different levels of services for clients with different needs. For example, MPEG-encoded video streams can be adapted for clients with lower bandwidth by dropping B-frames. Audio clients with limited resources can trade off bandwidth and fidelity with different compression mechanisms. These techniques can result in significant usability improvements by adapting the demands of the service to match the needs of the client and the available resources.

Traditionally, service adaptations like those for video delivery have been achieved via service negotiation built into the service-specific delivery protocols. However, this approach has the disadvantage of complicating the design and implementation of those protocols. More importantly, building service adaptations into the protocol implies design-time decisions on the types of adaptations which the client can negotiate, essentially requiring the service implementor to anticipate and provide for the needs of future clients. For example, consider a data stream that consists of stock trade information. Protocol designers might reasonably anticipate that a client would want to limit his data stream to information about only a few stocks, rather than reporting all trades. The protocol *could* require that the server send information indiscriminantly, forcing the client to discard that which he is not interested in. However, giving the client the ability to tailor his data flow would result in significant bandwidth savings and reduce the computational demands on both sides. As a result, support for basic tailoring is likely to be included in the protocol. However, what if the client only wanted information on stocks which were trading outside of a certain range? Or those which showed a certain percentage increase above a particular trading volume? What if the client was willing to accept degraded service and be sent only the 5-minute moving average instead of a report on every trade? None of these conditions are computationally complex, but as supporting every conceivable condition is impractical, protocol designers must

*This work was supported in part by NCSA/NSF grant C36-W63, NSF equipment grants CDA-9501637, CDA-9422033, and ECS-9411846, equipement donations from SUN and Intel Corporations, and NSF grant ASC-9720167.

"draw a line in the sand" demarking what they will support. Clients with more specific needs will be left with less than optimal data streams, resulting in wasted resources.

Situations like those described for the provision of stock data have two principal effects: (1) the waste of resources impacts the number of clients that can be serviced, and (2) limited options are presented to clients who find that they are not receiving adequate quality of service. Conversely, a highly tailorable service would allow an unhappy client to negotiate a lower level of service that might meet its most critical needs even in highly resource constrained environments.

The stock example is representative of a variety of real-world situations that involve the generation and distribution of data such that end users needs are met. Another example with which we have considerable familiarity is in the domain of scientific visualization. Consider a large parallel climate simulation running on a cluster machine. Such a simulation is capable of generating large amounts of state information on a continuous basis, and this state information is invaluable for scientists monitoring the progress of the simulation. Yet not every display program needs or is capable of rendering all of this data, and in response to this fact, we should not require the simulation to be tailored to meet the needs of every possible display program. A client-tailorable streaming data service is an appropriate solution to this problem.

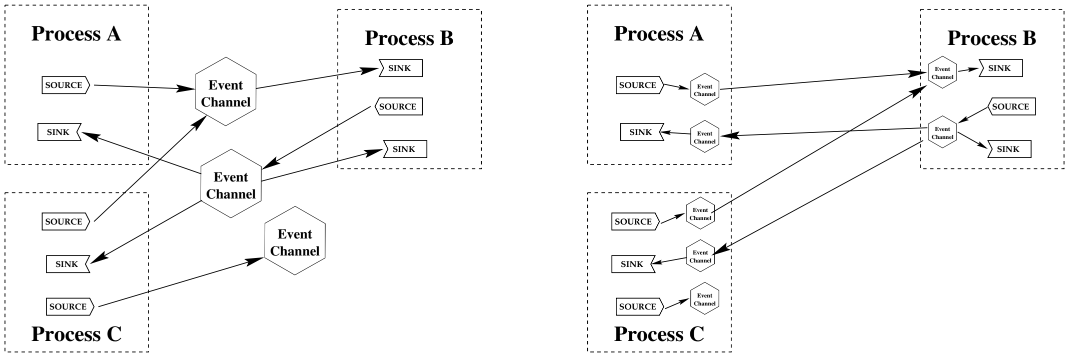
Well-designed middleware can play a key role in supporting the creation of effective methods for runtime stream adaptation, blurring the line between protocols and application-level adaptation[23]. Our contribution in this domain is ECho, an efficient event-based middleware designed to optimize the delivery of streaming data. ECho contains unique features to address the service adaptation difficulties raised in the previous paragraphs. In particular, ECho allows clients to *dynamically extend* servers with code fragments to customize their service. It employs dynamic code generation to ensure that these fragments can be executed efficiently, without the overheads of an interpreted language. It also allows clients to modify the behavior of these code fragments by updating *parameters* associated with the fragments. In addition the ECho system emphasizes three other characteristics of importance for Internet services:

Interoperability – ECho avoids assumptions about the architectures and operating systems hosting clients and servers. It runs transparently across a variety of Unix and Windows platforms.

Performance – ECho utilizes natural data representation (NDR) as a transmission format and uses a non-centralized event distribution scheme, thereby yielding data delivery overheads comparable to raw network times.

Evolvability – Service endpoints in ECho offer semantics that support transparent type extension and data type discovery. This allows service data contents to evolve over time while minimizing the risk to existing clients.

Section 2 of this paper describes ECho's basic functionality and its ability to specialize information flows with *derived event channels*. Section 3 briefly characterizes the performance of ECho's basic event delivery, then compares it to other infrastructures commonly used to implement streaming data systems. In particular, we compare ECho's basic latency and delivered bandwidth with that of CORBA event channels, Java's Jini distributed events, and an XML-based communication scheme. Like ECho, each of these is a higher-level delivery scheme that offers evolvability through some form of type extension or data type discovery. In addition, we include performance measures of ECho compared to MPI, a popular messaging system in the scientific computing community, to indicate the performance that might be expected from a data stream without the meta-data overhead of the other systems. Section 3 continues by examining the performance characteristics of our service specialization mechanism in several simple examples. Additionally, because Java-based mechanisms offer the most likely alternative implementations for specializing data streams, we present microbenchmarks for our approach and compare them with what might be achieved in a Java-based approach. Finally, Section 4 discusses some key areas of future work and summarizes our conclusions.



(a) Abstract view of Event Channels. (b) ECho Realization of Event Channels.

Figure 1: Using Event Channels for Communication.

2 ECho Functionality

ECho is an event-delivery system. Unlike some event systems whose performance characteristics are such that they are principally useful for control and notification purposes, ECho is designed as a high-performance data transport mechanism for both intra-process and network communication. While ECho's ability to transparently support both local and remote clients is useful in many circumstances, this paper will concentrate on ECho's role as a transport for network streaming data, where elements in the data stream map to data carried by ECho events.

ECho shares semantics common to a class of event delivery systems that use *channel-based subscriptions*. That is, an *event channel* is the mechanism through which event sinks and sources are matched. Source clients submit events to a specific channel, and only the sink clients subscribed to that channel are notified of the event. Essentially, channels are the entities through which the extent of event propagation is controlled. The CORBA Event Service[10] is also channel-based, with channels being distributed objects.

2.1 Efficient Event Propagation

Unlike many CORBA event implementations and other event services such as Elvin[21], ECho event channels are not centralized. Instead, channels are light-weight virtual entities. Figure 1A depicts a set of processes communicating using event channels. The event channels are shown as existing in the space between processes, but in their ECho realization, depicted in Figure 1B, they are distributed entities with bookkeeping data residing in each process where they are referenced. As a result, event messages are always sent directly from an event source to all sinks. Furthermore, by aggregating multiple event channels' communications into a single event stream linking two communicating address spaces, traffic for individual channels is multiplexed over shared communications links. These mappings are achieved by channels *created* once by some process and *opened* anywhere else they are used. The process creating the 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 the contact information for the creating process (as well as information identifying the specific channel).

ECho is implemented on top of DataExchange[6] and PBIO[3], packages developed at Georgia Tech to simplify connection management and heterogeneous binary data transfer. As such, it inherits from these packages its portability to different network transport layers and threads packages. DataExchange and PBIO operate across the various versions of Unix and Windows NT, have been used over the TCP/IP, UDP, and ATM communication protocols and across both standard and specialized network links.

Event channels provide the most straightforward mechanism through which a server application can segregate data according to its clients' anticipated needs. For example, in the stock tracking example described in Section 1, an event channel might be created for each individual stock, and trade information for that stock is submitted only to that channel. Clients desiring all trade information could subscribe to all channels, but clients interested in only a subset of stocks can limit their data stream by subscribing only to the appropriate channels. Of course, this approach can only be used for subdivisions of data that the service designer can reasonably anticipate. ECho allows clients with more specific needs to further refine these event streams with derived event channels, as described below in Section 2.3.

2.2 Event Types and Typed Channels

One of the differentiating characteristics of ECho is its support for efficient transmission and handling of fully typed events. Types may be associated with event channels, sinks and sources, and PBIO uses these types to automatically handle heterogeneous data transfer issues. Building this functionality into ECho allows for efficient layering that nearly eliminates data copies during marshalling and unmarshalling. As others have noted[15], careful layering to minimize data copies is critical to delivering full network bandwidth to higher levels of software abstraction. The layering with PBIO is a key feature of ECho that makes it suitable for applications which demand high performance for large amounts of data.

Base Type Handling and Optimization Functionally, ECho event types are most similar to user-defined types in MPI. The main differences are in expressive power and implementation. As with MPI, ECho event types describe C-style structures made up of atomic data types, and both systems support nested structures and statically-sized arrays. However, ECho's type system also supports null-terminated strings and dynamically sized arrays.[1]

While fully declaring message types to the underlying communication system gives the system the opportunity to optimize their transport, MPI implementations typically do not exploit this opportunity and often transport user-defined types even more slowly than messages directly marshalled by the application. In contrast, ECho and PBIO achieve a performance advantage by avoiding XDR, IIOP or other 'wire' representations different than the native representation of the data type. Instead, the wire format used is equivalent to the natural data representation (NDR) of the sender. Conversion to the native representation of the receiver is done upon receipt with dynamically generated conversion routines. As shown by the measurements in [5], PBIO 'encode' times do not vary with data size, and 'decode' times are much faster than MPI's. Because as much as two-thirds of the latency in a heterogeneous message exchange is software conversion overhead[5], PBIO's NDR approach yields round-trip message latencies as low as 40% of that of MPI's.

Type Extension ECho supports the robust evolution of sets of programs communicating with events, by allowing variation in data types associated with a single channel. In particular, an event source may submit an event whose type is a superset of the event type associated with its channel. Conversely, an event sink may have a type that is a subset of the event type associated with its channel. Essentially this allows a new field to be added to an event at the source without invalidating existing event receivers. This functionality can be valuable when a system evolves because it means that event contents can be changed without the need to simultaneously upgrade every component to accommodate the new type. ECho even allows type variation in intra-process communication, imposing no conversions when source and sink use identical types but performing the necessary transformations when source and sink types differ in content or layout.

The type variation allowed in ECho differs from that supported by message passing systems and intra-address space event systems. For example, the Spin event system supports only statically typed events[2]. Similarly, MPI's user defined type interfaces do not offer any mechanisms through which a program can interpret a message without *a priori* knowledge of its contents. Additionally, MPI performs strict type matching on message sends and receives, specifically prohibiting the type variation that ECho allows.

[1] In the case of dynamically sized arrays, the array size is given by an integer-typed field in the record. Full information about the types supported by ECho and PBIO can be found in [3].

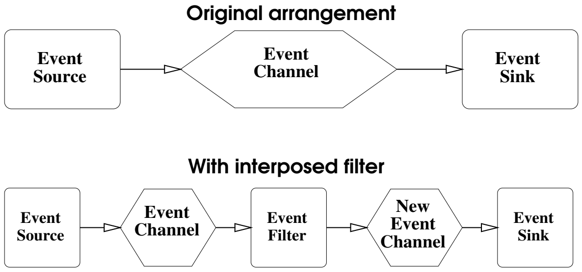


Figure 2: Source and sink with interposed event filter.

In terms of the flexibility offered to applications, ECho's features most closely resemble the features of systems that support the marshalling of objects as messages. In these systems, subclassing and type extension provide support for robust system evolution that is substantively similar to that provided by ECho's type variation. However, object-based marshalling often suffers from prohibitively poor performance. ECho's strength is that it maintains the application integration advantages of object-based systems while significantly outperforming them. As the measurements in Section 3 will show, ECho also outperforms more traditional message-passing systems in many circumstances.

2.3 Derived Event Channels

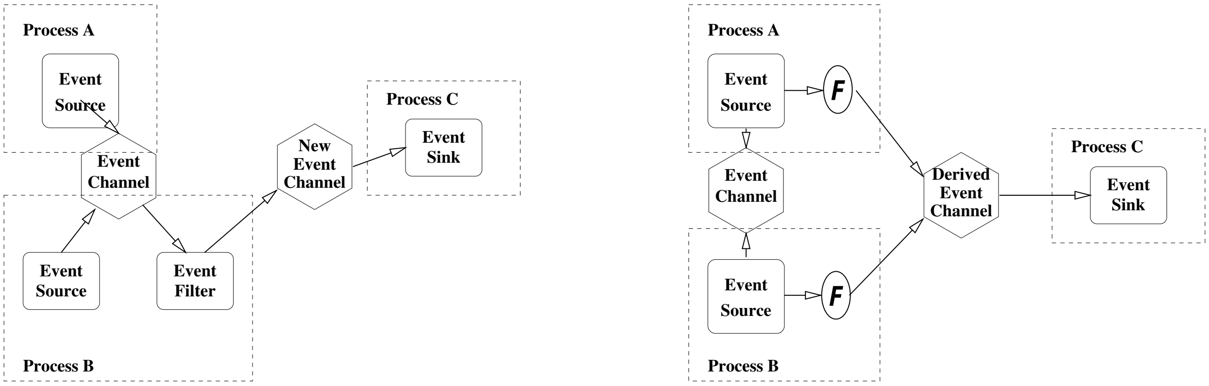
ECho's principal contribution to specializing data flows is the concept and realization of *derived event channels*. While it bears some similarity to prior work on content-based filtering (as in Siena[1] and Elvin[21]) and pattern-based filter/transformation (as in Gryphon[22]), ECho allows more general computations over event data and accomplishes those computations efficiently. This efficiency is based on semantics that don't require centralized event distribution and the use of dynamic code generation to create native filter/transformation functions. Our work is complementary with that of the Gryphon project as we are not looking at the optimal mapping of an information flow graph onto a network of brokers but rather concern ourself with the most efficient execution of such computations. The Java-based approach of DACE[9] offers broad generality in content-based subscriptions, but lacks the transformation capacity of derived event channels and offers significantly lower throughput and high latency than ECho. Our approach and implementation of ECho's derived event channel facility is described below.

2.3.1 General Model

Consider the situation where an event channel client is not really interested in *every* event submitted, but only wants every Nth event, or every event where a particular value in the data exceeds some threshold. Much computational and network overhead could be avoided if only the events of interest were transmitted. One way to approach this problem is to create a new event channel and interpose an event filter as shown in Figure 2.

The event filter can be located on the same node as the event source and is a normal event sink to the original event channel and a normal source to the new, or filtered, event channel. This solution does not disturb the normal function of the original event channel. However, it fails if there is more than one event source associated with the original event channel. The difficulty is that, as a normal sink, the event filter must live in some specific process. If multiple sources have subscribed to the original event channel and those sources are not co-located, as shown in Figure 3A, then there are still raw, unfiltered events traveling over the network from Process A to Process B.

The normal semantics of event delivery schemes do not offer an appropriate solution to the event filtering problem. Yet it is an important problem to solve because of the great potential for reducing resource requirements if unwanted events can be suppressed. Our approach involves extending event channels with the concept of a *derived* event channel. Rather than explicitly creating new event channels with intervening filter objects, applications that wish to receive



(a) Filter with more than one source

(b) A derived event channel and function F moved to event sources

Figure 3: Channel configurations.

filtered event data create a new channel whose contents are derived from the contents of an existing channel through an application-supplied derivation function, F. The event channel implementation will move the derivation function F to all event sources in the original channel, execute it locally whenever events are submitted, and transmit any resulting event via the derived channel. This approach has the advantage of eliminating unwanted event traffic and the associated waste of computational and network resources. In fact, if the derived event channel has sinks that are local to any of the sources in the original traffic, network traffic between those elements is avoided entirely. Figure 3B shows the logical configuration of a derived event channel.

2.3.2 Mobile Functions and the E-code Language

A critical issue in the implementation of derived event channels is the nature of the function F and its specification. Since F 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 F, such as to preselected values or to boolean operators,
- relying on pre-generated shared object files, or
- using interpreted code.

Having a relatively restricted filter language, such as one limited to combinations of boolean operators, is the approach chosen in the CORBA Notification Services[11] and in Siena[1]. This approach facilitates efficient interpretation, but the restricted language may not be able to express the full range of conditions useful to an application, thus limiting its applicability. To avoid this limitation, it is desirable to express F in the form of a more general programming language. One might consider supplying F in the form of a shared object file that could be dynamically linked into the process of the event source. Using shared objects allows F to be a general function, but requires the client to supply F as a native object file for each source. 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 F in an interpreted language, such as a TCL function or Java code. This would allow general functions and alleviate the difficulties with heterogeneity, but it would impact efficiency. Because of our intended application in the area of high performance computing, and because

```
{
    if ((input.trade_price < 75.5) || (input.trade_price > 78.5)) {
        return 1; /* submit event into derived channel */
    }
    return 0; /* do not submit event into derived channel */
}
```

Figure 4: A specialization filter that passes only stock trades outside a pre-defined range.

many useful filter and transformation functions are quite simple, we choose a different approach that maintains the highest efficiency. However, we consider interpreted languages as a complementary approach which has a place in less demanding applications.

The approach taken in ECho preserves the expressiveness of a general programming language and the efficiency of shared objects while retaining the generality of interpreted languages. The function F is expressed in E-Code, a subset of a general procedural language, and dynamic code generation is used to create a native version of F on the source host. E-Code 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. Extensions to other language features are straightforward and several are under consideration as described in Section 4.

E-Code's dynamic code generation capabilities are based on Icode, an internal interface developed at MIT as part of the 'C project[18]. Icode is itself based on Vcode[8], also developed at MIT by Dawson Engler. Vcode supports dynamic code generation for MIPS, Alpha and Sparc processors. We have extended it to support MIPS n32 and 64-bit ABIs, Sparc 64-bit ABI, and x86 processors[2]. Vcode offers a virtual RISC instruction set for dynamic code generation. The Icode layer adds register allocation and assignment. E-Code consists primarily of a lexer, parser, semanticizer and code generator.

ECho currently supports derived event channels that use E-Code in two ways. In the first, the event type in the derived channel is the same as that of the channel from which it is derived (the parent channel). In this case, the E-Code required is a boolean filter function accepting a single parameter, which is the input event. If the function returns a non-zero value, it is submitted to the derived event channel, otherwise it is filtered out. Event filters may be quite simple, such as the example in Figure 4. Applied in the context of our stock trading example, this filter passes trade information into the derived event channel only when the stock is trading outside of a specified range. When used to derive a channel, this code is transported in string form to the event sources associated with the parent channel, where it is parsed and native code is generated. The implicit context in which this code evaluated is a function declaration of the form:

`int f(`$\langle$*input event type*$\rangle$` input)`

where $\langle$*input event type*$\rangle$ is the type associated with the parent channel.[3] Once derived, the created channel behaves as a normal channel with respect to sinks. It has all of the sources of the parent channel as implicit sources, but new sources providing unfiltered events could also be associated with it.

While this basic support for event filtering is a very powerful mechanism for suppressing unwanted events in an data stream, ECho also supports derived event channels where the event types associated with the derived channel are not the same as that of the parent channel. In this case, E-Code is evaluated in the context of a function declaration of the form:

`int f(`$\langle$*input event type*$\rangle$` input, `$\langle$*output event type*$\rangle$` output)`

The return value continues to specify whether or not the event is to be submitted into the derived channel, but the differentiation between input and output events allows a new range of processing to be migrated to event sources.

[2] Integer x86 support was developed at MIT. We extended Vcode to support the x86 floating point instruction set (only when used with Icode).
[3] Since event types are required, new channels can only be derived from typed channels.

```
{
    int i;
    int j;
    double sum = 0.0;
    for(i = 0; i<37; i= i+1) {
        for(j = 0; j<253; j=j+1) {
            sum = sum + input.wind_velocity[j][i];
        }
    }
    output.average_velocity = sum / (37 * 253);
    return 1;
}
```

Figure 5: A specialization filter that computes the average of an input array and passes the average to its output.

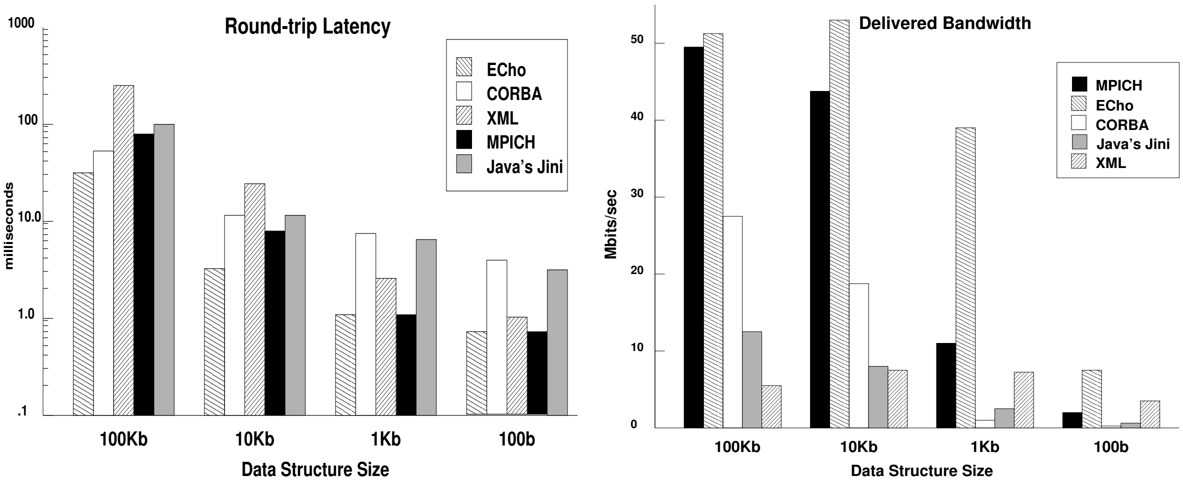
One use for this capability is remote data reduction. For example, consider event channels used for monitoring of scientific calculations, such as the global climate model described in [14]. Further, consider a client that may be interested in some property of the monitored data, such as an average value over the range of data. Instead of requiring the client to receive the entire event and do its own data reduction, considerable network resources could be saved by just sending the average instead of the entire event data. This can be accomplished by deriving a channel using a function which performs the appropriate data reduction. For example, the E-Code function defined in Figure 5 performs such an average over atmospheric data generated by the atmospheric simulation described in [14], thereby reducing the amount of data to be transmitted by nearly four orders of magnitude.

Static Variables and Parameterized Filters While the examples above are pure functions which have no scope beyond their input and output parameters, the full functionality offered by derived event channels is somewhat richer. The additional features do not fundamentally affect the nature of service specialization offered by derived event channels, but they do extend the range of possible specializations. One such feature is the ability to label variables as "static", so that their values persist across multiple invocations of the filter function. This allows the implementation of filters that require some amount of persistent state, such as a moving average computation, or a filter that simply passes every second or third stream element. However, as noted above, in the situation where the original channel has sources in multiple locations, each source gets its own copy of the derivation function *and of its persistent state.* Thus, a filter that passes every other event will end up delivering every other element *from each source*, rather than every other event submitted to the channel as a whole.

Another useful feature is the ability to *parameterize* a derivation function. Parameterization allows a parameter block to be associated with the derivation function. This block is read-only to the derivation function, but can be updated remotely in a push-type operation. This is useful in situations where a client needs to adapt the service with even finer granularity. For example, in the range filter in Figure 4, the range itself is specified as constants in the body of the filter. Yet it is clearly conceivable that a client might want to change the range over time. One could periodically destroy and re-derive channels with updated filter functions, but parameterized derivations allow a more straight-forward approach. As shown in Figure 6, the parameter block is used to hold the high and low bounds of the range. This allows the client to effect fine-grain adaptations to changing needs with relatively low overhead.

```
{
    if ((input.trade_price < param.range_low_bound) ||
        (input.trade_price > param.range_high_bound)) {
        return 1; /* submit event into derived channel */
    }
    return 0; /* do not submit event into derived channel */
}
```

Figure 6: Specialization filter range represented by parameterization data.



(a) Latency Comparison

(b) Bandwidth Comparison

Figure 7: Comparison with other event infrastructures

3 ECho Performance

3.1 Basic Data Streaming

In order to establish ECho's basic usability as a transport mechanism for streaming data, we first compare its performance with other transport mechanisms which might be used to implement a streaming data system. In particular, we compare ECho's basic latency and delivered bandwidth with that of CORBA event channels, Java's Jini distributed events, and an XML-based communication scheme. Like ECho, these are higher-level delivery schemes that can offer evolvability through some form of type extension or data type discovery. In addition, we include performance measures of MPI, a popular messaging system in the scientific computing community, to indicate the performance that might be expected from a data stream without the meta-data overhead of the other systems.

The precise performance characteristics of these packages depend upon such a wide variety of variables that we could not hope to characterize the entire performance space. Instead, we take an illustrative example and examine it. In this case, we present performance characteristics for a situation that might be typical of the scientific visualization example of Section 1 and use data from a mechanical engineering simulation of the effects of micro-structural properties on solid-body behavior. The data stream in this case consists of 100Kb state records, which we stream between a Sun Sparc and an x86-based PC over 100Mbps ethernet.[4] This situation might occur when a data stream from a cluster-based simulation is directed to a nearby host for visualization.

Figures 7A and 7B show basic event latency and bandwidth for several middleware systems in this sample environment. To show how the performance characteristics might vary with message size, we subset the original 100Kb state records and present measurements for 10Kb, 1Kb and 100b records, as well. The measurements cover event data exchange with ECho, an XML-based event delivery scheme, a CORBA event channel (using the ORBacus system), a Java-based scheme (making use of Jini Distributed Events) and our baseline MPICH.

Figure 7A shows that ECho's round-trip event latency is about half that of the other systems. While a more detailed discussion of the causes of the performance differences is beyond the scope of this paper, we can briefly characterize their nature. In latency measures, XML is hampered by a very high up/down conversion overhead and a bulky wire representation (ASCII). CORBA's latency is adversely affected by a centralized event distribution scheme

[4] The Sun machine is an Ultra 30 with a 247 MHz cpu running Solaris 7. The x86 machine is a 450 MHz Pentium II, also running Solaris 7.

that assigns the event channel object to a separate process. The figure shows that ECho actually improves on MPI's latency performance in most cases. This is largely a result of MPI's reliance on XDR as a wire format, which results in more data copying than is required in ECho.

Figure 7B compares the bandwidth delivered by each system relative to the original (unencoded) data size. It shows that ECho matches MPI's delivered bandwidth at large message sizes and performs considerably better for smaller sizes. Again, the extra data copying performed in MPICH plays a significant role in these results. CORBA maintains reasonable bandwidth for large message sizes, but the additional intervening process significantly affects bandwidth for smaller sizes. XML is greatly handicapped in any measure in which it is used as a wire format for transmitting binary data. The ASCII-based XML representation of a binary data structure is often five to seven times larger than the original data structure, wasting a significant amount of bandwidth.

A more complete discussion analyzing the causes of the performance differences in more detail and also presenting an analysis of the behavior of these communications systems in the context of application evolution and type extension can be found in [4].

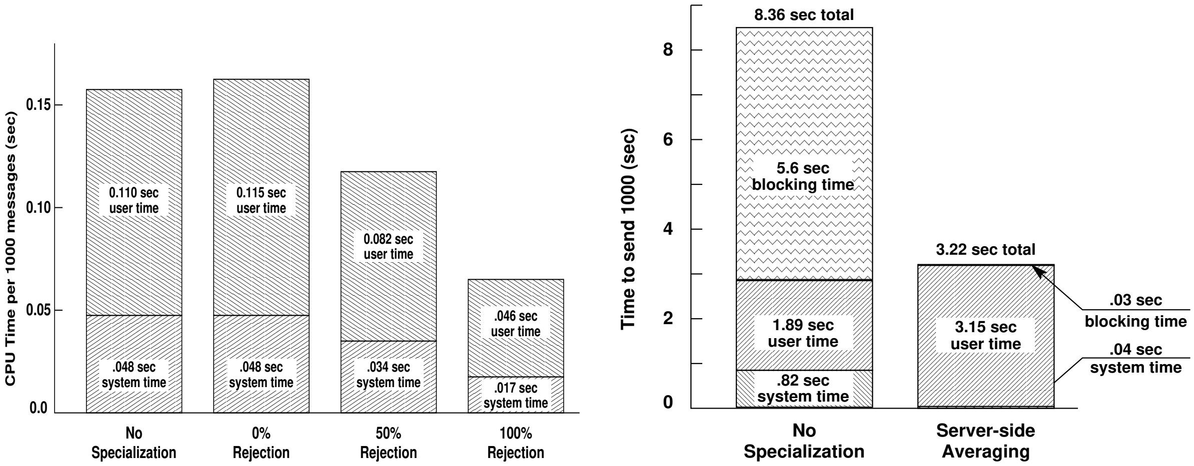
3.2 Effects of Stream Specialization

The primary benefit of stream specialization is the elimination of wasteful message traffic and the subsequent reduction of bandwidth requirements. As such, many of the benefits that are achievable through such specialization vary significantly with the extent of the unnecessary traffic. Gains achieved with ECho's event filtering approach are such that the resulting decrease in message traffic is directly proportional to the rejection rate of the specialization filter. Understanding such gains is a relatively straightforward exercise when ignoring issues like server, client, or network contention. For instance, if one half of the data streaming to a client is unwanted, then specialization allows him to receive twice the amount of **useful** information with the same bandwidth. If all the clients of a server exploit specialization to reduce their bandwidth requirements by one third, then the server can support 50% more clients at the same level of service it offered previously.

However, with the gains in reduced network usage come additional costs in the form of generating and executing specialization functions in the derived event channel. Those costs vary with the nature of the filter code, but interestingly, there are secondary, sometimes important effects, such as the elimination of unnecessary protocol stack execution, so that additional filter code execution may actually result in less total code executed by a server. Therefore, it is informative to examine the costs of filtering for representative examples.

First consider the range filter examples in Figure 4, taken from the context of our stock trading example and representing a simple range function on the stock's trade price. To realize this filter, ECho's dynamic code generation facility requires 4 milliseconds on a Sun Sparc Ultra 30 to create a native subroutine implementing the filter. The generated filter subroutine contains 27 Sparc instructions, and it executes in about 165 nanoseconds. In contrast, ECho's 7.5Mbps transfer rate for small (100 byte) messages implies that the server is spending about 107 microseconds to send each message. In other words, in this case, it is 'cheaper' to execute the filter than it is to send an unneeded message. Consequently, contrary to intuition, server load may be decreased, not increased, by the servers runtime, client-based extension with filters. In comparison, the same filter function implemented in Java, the most likely alternative representation of specialization functions, requires 1.8 microseconds for execution with Just-In-Time compilation enabled (3.7 microseconds otherwise). Therefore, arguments concerning the reduction vs. the increase in server load by their extension with filters do not trivially translate from ECho to other systems and from simple filters like this example to more complex filter functions.

Because service specialization potentially increases the computational load of the server, ECho's realization of specialization functions as dynamically generated native code plays a key role in making service specialization an attractive option. In particular, consider the relationship between the filter execution speed and the time required to send a message given in the previous paragraph. If the filter does not reject a particular message, the time required to execute the server function is a wasted resource over and above what it would have required to send the message on an unfiltered stream. However, if the message is rejected by the filter, the server saves the time it would have spent



(a) Server CPU utilization under different specialization scenarios

(b) Breakdown of times to send 1000 data elements with and without averaging specialization

Figure 8: Specialization performance information

sending the message. Thus, the ratio between the filter execution time and the time required to send the message on the unfiltered channel determines the break-even point of the specialization filter. A low ratio means that even if a filter has a relatively poor rejection ratio, it can still be computationally cheaper for the server do the filtering than it would have been to simply send the unfiltered messages. In the case of the sample filter operating on an event stream with 100 byte records, if the filter eliminates as little as 1 event in 30 it is still computationally cheaper for the server than the unfiltered situation.

Figure 8A illustrates the cost/benefits of filtering graphically. It shows the CPU utilization of a server sending a data stream to a single client. In this case, the data stream consists of 100 byte records generated at a particular rate. The figure shows the server's CPU utilization for four specialization scenarios: (1) an unspecialized stream, (2) a specialized stream in which no events are rejected, (3) a specialized stream in which 50% of the events are rejected, and (4) a specialized stream in which all events are rejected. Stream specialization obviously adds to the server's computational load in the form of additional overheads when none of the data stream is rejected. However, the overhead is slight and is quickly recovered if the specialization filter rejects a significant fraction of the data stream. At higher rejection rates, the use of a specialized data stream represents a cost savings to the server, despite the need to execute the filter function for every event.

The second example filter in Section 1 (Figure 5) is taken from the scientific visualization domain. A client might use this specialization function if it was only interested in the average of a large array in a data stream, as typically necessary when downsampling a data stream for visual rendering on variable-quality displays. This function is somewhat more complex than the first example. It requires 5.5 msecs for code generation and 1.28 msecs to execute. Furthermore, unlike the first example, this filter never rejects a data element, but merely transforms it by taking an average and forwarding the average instead. As a result, resource savings are due to reductions in required network processing at the server and in the reduction of required network bandwidth, important considerations for remote data visualization[12]. By comparison, the same filter function implemented in Java requires 13.33 msecs for execution with Just-In-Time compilation enabled, (75.43 msecs otherwise). This indicates that Java-based specialization may have such a costly run time that server extension would no longer be a win-win situation, *i.e.*, the increase in server computation cost might not be recouped in reduced network costs.

Figure 8B examines the impact of this sort of server adaptation. In particular, it shows a breakdown of the server's time costs in sending a client 1000 data elements. If server adaptation is not available, the server must send the client the unprocessed array, with costs represented by the bar labeled “No Specialization.” Alternatively, if derived event channels are used to specialize the service, the server performs the averaging computation and experiences costs represented by the bar labeled “Server-side Averaging.” Obviously, specializing the service significantly increases the “user cpu time” component of the costs, but this is more than offset by the dramatic reduction in system and network blocking costs which result from the decrease in message size.

4 Conclusions and Future Work

This paper examined ECho, an event-based middleware designed to optimize the delivery of streaming data. The ECho system provides a publish-subscribe communication model that supports efficient event delivery in heterogeneous systems, runtime type extension, and type discovery. More importantly, ECho provides an innovative support for runtime stream specialization through a mechanism called *derived event channels*. This mechanism allows clients to dynamically extend servers with code fragments to tailor data streams to suit their needs.

The performance evaluation in Section 3 established two salient points. First, while object-based systems provide functionality that is similar to the type extension and type discovery features of ECho, ECho does it with significantly better performance, both in terms of delivered bandwidth and end-to-end latency. In fact, the measurements in Section 3.1 show that ECho matches and, in most cases, outperforms MPI in both metrics, thereby demonstrating performance that outpaces that of a commonly accepted high-performance communication paradigm.

Secondly, the evaluation in Section 3.2 shows that data stream specialization can be a win-win situation for servers and clients, reducing network traffic and lowering the computational costs for both sides. Stream specialization naturally reduces the amount of network traffic and the client's computational load by moving client-side computations to the server. In a obvious result, the timings presented here show that while one portion of the server's costs are necessarily increased by the need to run the client-provided specialization code, it is not unusual to recoup the costs through through other savings. In particular, the reduced network traffic that comes with stream specialization means that the server spends less time in network transmission. ECho's dynamically code generated specialization functions are sufficiently fast that filters need only reduce the data stream by a relatively small fraction to repay their costs. Alternative implementation mechanisms based on interpreted languages would imply significantly higher server-side costs, making stream specialization beneficial in fewer situations.

While this paper demonstrates initial results with the deployment of ECho and its notion of derived event channels, as we are gaining experiences with using them, we are adding to the system several key notions that will enable us to create the future, ubiquitous, stream-based applications we are targeting with our work. A few of these extensions are described next.

Visibility into Source Process [7] anticipates using the functionality offered by derived event channels for program steering or more generally, for runtime adaptation of programs. Currently, external program steering typically requires monitoring events to be delivered to an external process. There they are interpreted and actions can be taken to affect the state of the application being steered. Like filter functions, steering decision functions may be as simple as comparing an incoming value with a threshold and steering the program if the threshold is exceeded. As a result, program steering requires a minimum of a network round-trip, and it is an asynchronous process. If the steering functions could be migrated into the steered application, much like we are moving filter functions to event sources, steering latencies could be reduced by several orders of magnitude and undertaken synchronously. When program steering is performed by a human-in-the-loop system, latency and synchronicity are of little importance. But some program steering, and much of program and protocol adaptation, can be performed algorithmically, with external code making the adaptation or steering decisions. Such adaptations must be rapid and often must be synchronous in order to be valid. This work holds significant potential in terms of allowing programs, system abstractions and even operating systems to adapt to changing usage and resource situations[20, 17].

However, doing program steering or program adaptation in a derivation function requires that the function have visibility into application program to call functions or affect values there. To accomplish these ends, we envision associating a “context” with an event channel. The context would specify program elements which might be accessible to derivation functions. In this way, visibility into the sending application would be controlled by the sender but the use and updating of visible elements would be specified by the receiver through derivation function.

Function Analysis There is also potential benefit in specializing and analyzing derivation functions themselves. In particular, an event source clearly has an interest in understanding the nature of the code it agrees to execute for a derivation function. While the simplest functions may be safe in the sense that they have no visibility into source's environment, they may walk off the end of event arrays or contain infinite loops. However, in this environment we know the size and extent of all data types. Therefore, we can easily generate array bounds checks and can also consider analyzing the specified functions for computational complexity and termination. An event source on a highly loaded machine might be given the option of rejecting a derivation that would increase its workload. More generally, since derived event channels always represent some tradeoff between compute and network resources, we can consider creating a more generalized mechanism through which tradeoff decisions can be made in the context of dynamic feedback from resource monitoring. Such a system could greatly ease the process of building applications which perform robustly in today's dynamic network environments.

Future applications being developed with ECho include those targeted by the Infosphere project[19], which include Continual Queries to web sites[16], ubiquitous applications via wireless communication media and involving embedded systems (*i.e.*, we are currently porting ECho to wireless-connected Linux laptops and to Palmtops), and applications in smart homes (*i.e.*, we are going to use ECho within Georgia Tech's Smart Home project[13] as the delivery medium for both data and control signals in home applications ranging from simple sensor data capture and analysis to the transport and analysis of image data streams.). ECho has also been used in a range of high performance applications, including a Hydrology Workbench used by collaborating scientists and a similar system used for collaboration across the Internet[12].

References

[1] Antonio Carzaniga, David S. Rosenblum, and Alexander L. Wolf. Design of a scalable event notification service: Interface and architecture. Technical report, Department of Computer Science, University of Colorado at Boulder, August 1998. Technical Report CU-CS-863-98.

[2] Craig Chambers, Susan J. Eggers, Joel Auslander, Matthai Philipose, Markus Mock, and Przemyslaw 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.

[3] Greg Eisenhauer. Portable self-describing binary data streams. Technical Report GIT-CC-94-45, College of Computing, Georgia Institute of Technology, 1994. *(anon. ftp from ftp.cc.gatech.edu)*.

[4] Greg Eisenhauer, Fabián E. Bustamante, and Karsten Schwan. Event services for high performance computing. To appear at High Performance Distributed Computing - HPDC' 2000.

[5] Greg Eisenhauer and Lynn K. Daley. Fast heterogeneous binary data interchange. In *Proceedings of the 9th Heterogeneous Computing Workshop (HCW 2000)*, pages 90–101, May 3-5 2000.

[6] Greg Eisenhauer, Beth Schroeder, and Karsten Schwan. Dataexchange: High performance communication in distributed laboratories. *Journal of Parallel Computing*, 24(12-13):1713–1733, November 1998.

[7] Greg Eisenhauer and Karsten Schwan. An object-based infrastructure for program monitoring and steering. In *Proceedings of the 2nd SIGMETRICS Symposium on Parallel and Distributed Tools (SPDT'98)*, pages 10–20, August 1998.

[8] Dawson R. Engler. Vcode: a retargetable, extensible, very fast dynamic code generation system. In *Proceedings of ACM SIGPLAN'96 Conference on Programming Language Design and Implementation*, 1996.

[9] Patrick Eugster, Rachid Guerraoui, and Joe Sventek. Distributed asynchronous collections: Abstractions for publish/subscribe interaction. In *the proceedings of ECOOP'00*. Springer Verlag (LNCS), June 2000.

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

[11] Object Management Group. Notification service. http://www.omg.org, Document telecom//8-01-01, 1998.

[12] Carsten Isert and Karsten Schwan. ACDS: Adapting computational data streams for high performance. In *Proceedings of International Parallel and Distributed Processing Symposium (IPDPS)*, May 2000.

[13] Cory D. Kidd, Robert J. Orr, Gregory D. Abowd, Christopher G. Atkeson, Irfan A. Essa, Blair MacIntyre, Elizabeth Mynatt, Thad E. Starner, and Wendy Newstetter. The aware home: A living laboratory for ubiquitous computing research. In *The Proceedings of the Second International Workshop on Cooperative Buildings - CoBuild'99*, October 1999. http://www.cc.gatech.edu/fce/house.

[14] Thomas Kindler, Karsten Schwan, Dilma Silva, Mary Trauner, and Fred Alyea. A parallel spectral model for atmospheric transport processes. *Concurrency: Practice and Experience*, 8(9):639–666, November 1996.

[15] Mario Lauria, Scott Pakin, and Andrew A. Chien. Efficient layering for high speed communication: Fast messages 2.x. In *Proceedings of the 7th High Performance Distributed Computing (HPDC7)*, July 1998.

[16] Ling Liu, Calton Pu, and Wei Tang. Continual queries for internet scale event-driven information delivery. *IEEE Transactions on Knowledge and Data Engineering, Special issue on Web Technologies*, 11(4):610–628, July/Aug 1999. http://www.cc.gatech.edu/projects/disl/CQ.

[17] B. Mukherjee and K. Schwan. Implementation of scalable blocking locks using an adaptive threads scheduler. In *International Parallel Processing Symposium (IPPS)*. IEEE, April 1996.

[18] Massimiliano Poletto, Dawson Engler, and M. Frans Kaashoek. tcc: A template-based compiler for `c. In *Proceedings of the First Workshop on Compiler Support for Systems Software (WCSSS)*, February 1996.

[19] Calton Pu. Infosphere – smart delivery of fresh information. http://www.cse.ogi.edu/sysl/projects/infosphere//

[20] Daniela Ivan Rosu, Karsten Schwan, Sudhakar Yalamanchili, and Rakesh Jha. On adaptive resource allocation for complex real-time applications. In *18th IEEE Real-Time Systems Symposium, San Francisco, CA*, pages 320–329. IEEE, Dec. 1997.

[21] Bill Segall and David Arnold. 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.

[22] Robert Strom, Guruduth Banavar, Tushar Chandra, Marc Kaplan, Kevan Miller, Bodhi Mukherjee, Daniel Sturman, and Michael Ward. Gryphon: An information flow based approach to message brokering. In *International Symposium on Software Reliability Engineering '98*, 1998.

[23] Jim Waldo. The end of protocols. http://developer.java.sun.com/developer/technicalArticles/jini/protocols.html.