

Event-based Systems: Opportunities and Challenges at Exascale

Greg Eisenhauer
College of Computing
Georgia Tech
eisen@cc.gatech.edu

Matthew Wolf^{*}
College of Computing
Georgia Tech
mwolf@cc.gatech.edu

Hasan Abbasi
College of Computing
Georgia Tech
habbasi@cc.gatech.edu

Karsten Schwan
College of Computing
Georgia Tech
schwan@cc.gatech.edu

ABSTRACT

Streaming data models have been shown to be useful in many applications requiring high-performance data exchange. Application-level overlay networks are a natural way to realize these applications' data flows and their internal computations, but existing middleware is not designed to scale to the data rates and low overhead computations necessary for the high performance domain. This paper describes EV-Path, a middleware infrastructure that supports the construction and management of overlay networks that can be customized both in topology and in the data manipulations being performed. Extending from a previous high-performance publish-subscribe system, EVPath not only provides for the low overhead movement and in-line processing of large data volumes, but also offers the flexibility needed to support the varied data flow and control needs of alternative higher-level streaming models. We explore some of the challenges of high performance event systems, including those experienced when operating an event infrastructure used to transport IO events at the scale of hundred+ thousand nodes. Specifically, when transporting output data from a large-scale simulation running on the ORNL Cray Jaguar petascale machine, a surprising new issue seen in experimentation at scale is the potential for strong perturbation of running applications from inappropriate speeds at which IO is performed. This requires the IO system's event transport to be explicitly scheduled to constrain resource competition, in addition to dynamically setting and changing the topologies of event delivery.

Categories and Subject Descriptors

C.4.4 [Communications Management]: Network Communication

^{*}joint, Oak Ridge National Laboratory, Oak Ridge TN

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

DEBS '09 Nashville, TN

Copyright 2009 ACM 978-1-60558-665-6/09/07....\$10.00.

Keywords

events, middleware, overlay, high-performance, communication

1. INTRODUCTION

Middleware plays an important role in bridging the high performance communications goals of the application level and the capabilities and capacities of the underlying physical layer. The publish-subscribe paradigm is well-suited to the reactive nature of many novel applications such as collaborative environments, compositions of high performance codes, or enterprise computing. Such applications often evolve from tightly-coupled components running in a single environment to collaborating components shared amongst diverse underlying computational centers. Traditional high performance scientific application components now may also include those responsible for data analysis, temporary and long term storage, data visualization, data preprocessing or staging for input or output. In the business domain, business intelligence flows may include scheduling (airline planning), logistics (package delivery), ROI estimation (just-in-time control), or rule engines (dynamic retail floor space planning). Such components may run on different operating systems and hardware platforms, and they may be written by different organizations in different languages. Complete "applications", then, are constructed by assembling these components in a plug-and-play fashion.

The *decoupled* approach to communication offered by event-based systems has been used to aid system adaptability, scalability, and fault-tolerance [13] as it enables the rapid and dynamic integration of legacy software into distributed systems, supports software reuse, facilitates software evolution, and has proven to be a good fit for component-based approaches. However, this work focuses on those applications and software stacks where *performance* is at least as critical as the other benefits of event-style communication.

In particular, high performance event systems can be categorized along two axes. One axis categorizes the event subscription model – ranging from category-based subscriptions (*e.g.*, all "passenger:boarding" events) at one end to complex event processing at the other (*e.g.*, event D only delivered if event A is in range $[a1 : a2]$, event B is in $[b1 : b2]$, and there has been no event C since time T_i). In other words, this

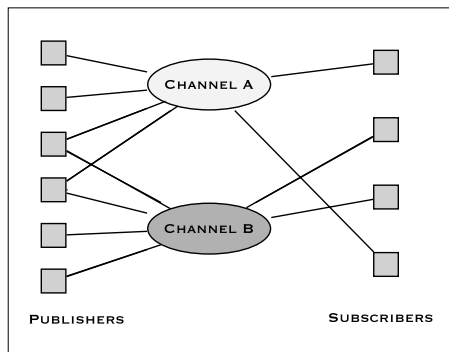


Figure 1: A conceptual schematic of a channel-based event delivery system. Logically, data flagged as belonging to channel A or B flows from the publishers through a channel holder and to the subscriber. Particular system implementations may or may not require a centralized channel message center.

axis of the design space defines whether delivery is based on metadata or data selection criteria.

This first axis is generally useful for characterizing any event system. The restriction for high performance is addressed in the second axis, with a trade-off between scalability (*e.g.* millions of publisher or subscribers) and throughput (*e.g.* handling 10k messages per second). Note that this is not to say that either of these axes are strict dichotomies – particular systems may in fact be able to handle both scalability and throughput, or deal with both metadata and complex data selection rules. However, the design choices made for most systems tend to be focused on one or the other, with very few exceptions. For example, channel-based systems like Figure 1 may be more amenable to scaling performance than centralized systems depicted in Figure 2. In the following section, we explore the design trade-offs along these axes in more detail by examining some exemplar event systems.

2. DESIGN POINTS

Historically, event-based publish/subscribe communication owes much to early work on group communication protocols in the distributed systems domain such as [3]. These early systems were largely concerned with abstract properties of group communication, such as fault tolerance, message ordering, and distributed consensus. As group communication grew in popularity a wide variety of systems were developed, each providing a different set of features and performance compromises. Some systems provide features that are incompatible with other potential performance goals. For example, fully-ordered communication requires a very heavyweight protocol that is unlikely to achieve high performance and may not scale to many subscribers. Because of these incompatibilities, each implementation tended to represent a single point in the possible design space which was intended to be optimal for the types of applications it was attempting to address.

TIBCO’s Information Bus [28] was an early commercial offering in the enterprise space used for the delivery of financial data to banks and similar institutions. TIBCO is an example of a throughput-oriented, *subject-* or *topic-based* sub-

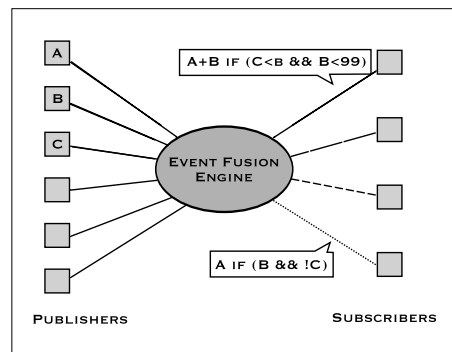


Figure 2: A conceptual schematic of a complex event processing delivery system. Logically all data flows into a centralized processing engine that runs each subscriber’s personalized subscription request.

scription system, where events are associated with a particular textual “topic” and publishing or subscribing to events associated with a topic constitutes joining a group communication domain. For TIBCO and many such systems, topics are hierarchical, so one subscribe to all NASDAQ trades by subscribing to “/equities/nasdaq/” or narrow the subscription to a specific company by subscribing to “/equities/nasdaq/IBM”. Because event routing in these systems depends only upon the relatively static topic information, topic-based systems usually don’t require transporting all events to a central location and are well suited for scaling to many subscribers. However, there is no opportunity for subscribers to request some subset of events other than the subdivisions predetermined by the event source by its choice of publishing topic.

Vitria’s Velocity server [26] and the CORBA Event Service[24] represent a design point that is similar in flexibility to the TIBCO solution, except that rather than being labeled with textual topics, events are published into “event channels”. These channels control the distribution of events so that subscribers associated themselves with a specific channel and receive events that are submitted to his channel. In CORBA, these channels are objects themselves, implying a certain centralization in event routing, but conceptually the *channel-based* model of event subscription shares much with the *topic-based* model in that one can be used to implement the other with great success. Thus, topic-based and channel-based systems share both the upside and downside of their relatively static event routing models: the potential for scalability, but also the limits of subscriber flexibility.

In *content-based* subscription systems, such as Sienna[7] and Gryphon[27], subscribers can express interest in events based upon certain run-time properties. Subscriptions are expressed as a boolean function over these properties and effectively deliver a filtered version of the overall event stream to the subscriber. Systems supporting content-based subscriptions vary widely in the nature of the properties and the means through which the filtering function is expressed, usually as a function of how event data itself is represented. In traditional message-style systems where the event data is simply a block of data to be moved, properties might be represented as a separate set of name/value pairs associated with the event. This approach is taken by the CORBA Noti-

fication Service[23], a variation of the CORBA event service that adds event filtering to its subscription system. Alternatively, if the message is self-describing or supports some kind of introspection on its content, the run-time properties available for filtering can be the data elements of the event itself. For example, an event representing trading data might contain elements such as the company name “IBM” and the trade price “87.54”.

At another point in the design space, grid computing allows the interconnection of heterogeneous resources to provide large scale computing resources for high performance applications. Because of its distributed focus, scalability of solutions is at a premium. The Open Grid Services Architecture (OGSA) [16] provides a web service interface to Grid systems. Two separate notification specifications are defined for event updates within the OGSA, WS-Notification [14] and WS-Eventing [4]. WS-Notification defines three interlinked standards, WS-BaseNotifications, WS-Brokered Notification and WS-Topics. A subscription is made to a specific notification producer, either directly for WS-Base Notification or through a broker for WS-BrokeredNotification. In addition, WS-Topics allows subscriptions to specific “topics” through the definition of a topic space. By using XML to describe the subscription and the event, the WS-Notification standard provides a high level of interoperability with other open standards. The downside to using XML for such processing is the performance implication of text parsing. Implementations of WS-Notification, such as WS-Messenger [18] trade-off event and query complexity in order to provide for a scalable high throughput event processing system. Similarly Meteor [19] is a content based middleware that provides scalability and high throughput for event processing. Meteor achieves scalability by refining the query into an aggregation tree using a mapping of a space filling curve. By aggressively routing and aggregating queries, Meteor can maintain low computational and messaging overheads.

In the quadrant addressing both scalability and complex event processing, Cayuga [5] is a general purpose stateful event processing system. Using an operator driven query language, Cayuga can match event patterns and safety conditions that traditional data stream languages cannot easily address. Cayuga defines a set of operations as part of its algebra allowing the implementation and optimization of complex queries for single events and event streams [9, 8]. This algebra defines a formal event language for Cayuga.

The ECho publish-subscribe system [10] provides a channel-based subscription model, but it also allows subscribers to customize, or *derive*, the publisher’s data before delivery. ECho was designed to scale to the high data rates of HPC applications by providing efficient binary transmission of event data with additional features that support runtime data-type discovery and enterprise-scale application evolution. ECho provides this advanced functionality while still delivering full network bandwidth to application-level communications, a critical capability for high-performance codes.

2.1 Next Generation Event Delivery.

In observing the space of high performance event systems, and in particular some of the authors’ experience in developing the ECho system, we derived a set of requirements that we used in designing EVPath, a next generation of high performance distributed event middleware.

Instead of offering publish/subscribe services directly, EV-

Path’s approach is to provide a basis for implementing diverse higher level event-based communication models achieving the following goals:

- *Separating event-handling from control.* Event systems are largely judged on how efficiently they transport and process events, while the control system may be less performance critical. EVPath’s API permits the separation of these two concerns, thereby making it easy to experiment with different control mechanisms and semantics while still preserving high performance event transport.
- *Using overlay networks.* EVPath allows the construction and on-the-fly reconfiguration of overlay networks with embedded processing. Such ‘computational overlays’ are the basis for any attempt to address scalability in event messaging.
- *Runtime protocol evolution.* For flexibility in event handling, protocols can evolve at runtime, as demonstrated with previous work on dynamic ‘message morphing’ [2]. This improves component interoperability through the runtime evolution of the types of events being transported and manipulated.
- *Efficient event filtering/transformation.* Basic event transport abstractions support the implementation of efficient event filtering/transformation, enabling some degree of trade-off between moving processing to events versus events to processing.
- *Open access to network resources.* EVPath provides a clean mechanism for exposing information about and access to the network at the application level, making it easier for applications to leverage network capabilities, such as by adjusting their event processing or filtering actions [6].
- *Supporting multiple styles of network transports.* Event transport can utilize point-to-point, multicast, or other communication methods, without ‘breaking’ the basic event semantics.

The outcome is a flexible event transport and processing infrastructure that serves as an efficient basis for implementing a wide range of higher level communication models [10, 20, 6, 29].

3. THE EVPATH ARCHITECTURE

In this section, we explore EVPath as a more in-depth case study of an event system architecture. As a design point, EVPath is an event processing architecture that supports high performance data streaming in overlay networks with internal data processing. EVPath is designed as an event transport middleware layer that allows for the easy implementation of overlay networks, with active data processing, routing, and management at all points in the overlay. EVPath specifically does not encompass global overlay creation, management or destruction functions. In essence, EVPath is designed as a new basis for building event messaging systems; it does not provide a complete event system itself.

At its core, EVPath implements the parts of the system that are directly involved in message transport, processing and delivery, leaving “management” functions to higher layers. Different implementations of those management layers might support different communications paradigms, management techniques or application domains, all exploiting the base mechanisms provided in EVPath. Because of this, EVPath is envisioned as an infrastructure that can encom-

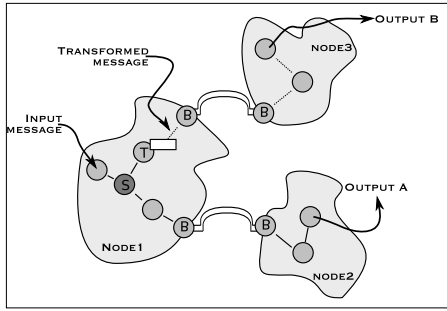


Figure 3: A conceptual schematic of an event delivery system built using the EVPath library

pass the full range of design points in the event system space.

EVPath represents an entire streaming computation, including data inflow, filtering, transformation, aggregation, and delivery as entities in a dataflow graph. Those entities can then be mapped by an externally provided management layer onto nodes and processes as appropriate given data locality, processing, and network resource demands, or throughput and availability SLAs. As demands and resource availability change, the computational elements and data flows can be changed by remapping elements or splitting flows to avoid network or computation bottlenecks.

EVPath is built around the concept of “stones” (like stepping stones) that are linked to build end-to-end “paths”. While the “path” is not an explicitly supported construct in EVPath (since the control of transport is key to any particular design solution), the goal is to provide all of the local-level support necessary to accomplish path setup, monitoring, management, modification, and tear-down. Here paths are more like forking, meandering garden paths than like the linear graph-theoretic concept.

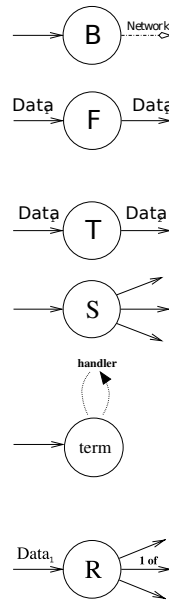
Stones in EVPath are lightweight entities. Stones of different types perform data filtering, data transformation, mux and demux of data, as well as the transmission of data between processes over network links. In order to support in-network data filtering and transformation, EVPath also is fully type-aware, marshalling and unmarshalling application messages as required for its operations. Further, as much as possible, EVPath tries to separate the specification of path connectivity (the graph structure) from the specification of operations to be performed on the data.

3.1 Taxonomy of Stone Types

Figure 4 describes some basic types of stones that EVPath makes available for data stream construction. Most of the design points we have discussed can be addressed just with these building blocks. As show in Figure 3, stones in the same process are linked together by directing the output of one stone to the input of another. Bridge stones transport events to specific stones in another process or on another host. Events may be submitted directly to any stone, may be passed from stone to stone locally, or may arrive from a network link. Event processing at a stone does not depend upon how the event arrived, but only upon the type of data and attributes the event is carrying, making the overlay transport publisher-blind as a result.

3.2 Data Type Handling

A key feature of many event systems is the handling of



A bridge stone is used for non-local propagation (network transmission) of events.

A filter stone runs a function ‘F’ on input events. If that function returns 0, the event is discarded. Otherwise, it is passed to the output link.

A transform stone is similar to the filter stone, except that the input and output data types are not necessarily the same.

A split stone takes incoming events and replicates them along each of its output links.

A terminal stone delivers incoming events to the application by passing event data as a parameter to an application-provided handler.

Like a split stone, a router stone has multiple outputs. But instead of duplicating incoming events on each of them, it runs a function ‘R’ with the event data as a parameter. If the functions returns ‘N’, the stone passes the event on to its N’t output link.

Figure 4: Basic stone types

```

type def struct {
    int cpu;
    int memory;
    double network;
} Msg, *MsgP;

FMField Msg_field[] = {
    {"load", "integer", sizeof(int), FMOffset(MsgP, load)},
    {"memory", "integer", sizeof(int), FMOffset(MsgP, memory)},
    {"network", "float", sizeof(double), FMOffset(MsgP, network)},
}

```

Figure 5: Sample EVPath data structure declaration

data between heterogeneous systems – whether by adopting a common format like XML or a common run-time like Java. Unlike systems that transport events as undifferentiated blocks of bytes, EVPath data manipulation is based upon fully-typed events processed natively on the hardware. Event data is submitted to EVPath marked up as a C-style structure, in a style similar to ECho[10] and PBIO[11]. Terminal stones pass similar structures to application-level message handlers, allowing different handlers to be specified for different possible incoming data types. Figure 5 shows a very simple structure and its EVPath declaration. EVPath events are structures which may contain simple fields of atomic types (as in Figure 5), NULL-terminated strings, static and variable-sized multi-dimensional arrays, substructures and pointers to any of those elements. EVPath structures can also describe recursive data structures such as trees, lists and graphs.

EVPath uses the FMField description to capture necessary information about application-level structures, including field names, types, sizes and offsets from the start of the structure. The type field, either “integer” or “float” in the example, encodes more complex field structures such as static and variable arrays (“integer[1000]”, “float[dimen1][dimen2]”), substructures and pointers to other items. In all cases, EVPath can marshal and unmarshal the data structures as

necessary for network data transmission, transparently handling differences in machine representation (*e.g.* endianness) and data layout.

In order to support high-performance applications, EVPath performs no data copies during marshalling. As in PBIO [11], data is represented on the network largely as it is in application memory, except that pointers are converted into integer offsets into the encoded message. On the receiving side, dynamic code generation is used to create customized unmarshalling routines that map each incoming data format to a native memory structure. The performance of the marshalling and unmarshalling routines is critical to satisfying the requirements of high-performance applications.

3.3 Mobile Functions and the Cod Language

A critical issue in the implementation of EVPath is the nature of the functions (F , T , *etc.*) used in the implementation of the stones described above. The goal of EVPath is to support a flexible and dynamic computational environment where stones might be created on remote nodes and possibly relocated during the course of the computation, as this feature is necessary to realize with scalability the complex data processing portion of the design space. In this environment, a pointer to a compiled-in function 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.

As noted in Section 2, having a relatively restricted filter language, such as one limited to combinations of boolean operators, is the approach chosen in other event systems, such as the CORBA Notification Services [23] and in Sienna [7]. 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 utility. 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 stone that required it; indeed, EVPath does support this mode if a user so chooses. Using shared objects allows F to be a general function but requires F to be available as a native object file everywhere it is required. This is relatively easy in a homogeneous system, but it becomes increasingly difficult as heterogeneity is introduced, particularly if semblance type safety is to be maintained.

In order to avoid problems with heterogeneity, one might supply F in an interpreted language, such as a TCL function or Java byte code. This would allow general functions and alleviate the difficulties with heterogeneity, but it potentially impacts efficiency. While Java is attractive as a unified solution because of its ability to marshall objects, transport byte-code and effectively execute it using just-in-time compilation, its use in event systems can be problematic. In particular, consider that traditional decoupling between event suppliers and consumers might restrict them from effectively sharing events as specific application-defined objects of a common class. Some solutions have been proposed[12], but at this point, Java-based marshalling of unbound objects is not the highest-performing messaging solution.

```

{
  if ((input.trade_price < 75.5) ||
      (input.trade_price > 78.5)) {
    return 1; /* pass event over output link */
  }
  return 0; /* discard event */
}

```

Figure 6: A specialization filter that passes only stock trades outside a pre-defined range.

Because of this, EVPath chose a different approach in order to achieve the highest event rate and throughput possible. In particular, EVPath’s approach 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 Cod (C On Demand), a subset of a general procedural language, and dynamic code generation is used to create a native version of F on the source host. Cod is a subset of C, supporting the C operators, **for** loops, **if** statements and **return** statements.

Cod’s dynamic code generation capabilities are based on the Georgia Tech DILL package that provides a virtual RISC instruction set. Cod consists primarily of a lexer, parser, semanticizer, and code generator. The Cod/DILL system generates native machine code directly into the application’s memory without reference to an external compiler. Only minimal optimizations are performed, but, because of the simplicity of the marshalling scheme (allowing direct memory-based access to event data), the resulting native code still significantly outperforms interpreted approaches.

Event filters may be quite simple, such as the example in Figure 6. Applied in the context of a stock trading example, this filter passes on trade information only when the stock is trading outside of a specified range. This filter function requires 330 microseconds to generate on a 2Ghz x86-64, comprises 40 instructions and executes in less than a microsecond. Transform stones extend this functionality in a straightforward way. For example, the Cod function defined in Figure 7 performs such an average over wind data generated by an atmospheric simulation application, thereby reducing the amount of data to be transmitted by nearly four orders of magnitude.

Event transformations like the one performed in Figure 7, are not achievable in typical enterprise-focused event systems that are mostly concerned with smaller-scale and simpler events. However, it is not atypical of the types of data

```

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

```

Figure 7: A specialization filter that computes the average of an input array and passes the average to its output.

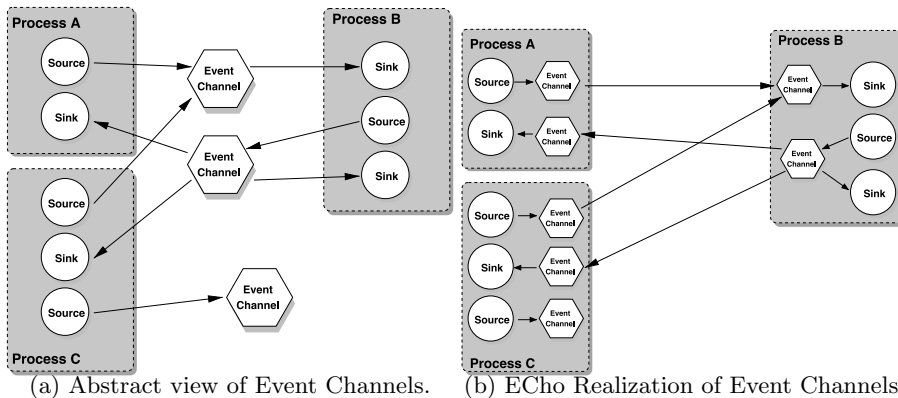


Figure 8: Using Event Channels for Communication.

processing concerns that might occur in scientific computing.

3.4 Meeting Next Generation Goals

EVPPath’s architecture is designed to satisfy the next generation goals outlined in Section 2.1. It encourages a separation of control and data transport functionality by providing a building block-like set of functional entities for constructing computational overlay networks. But it does not provide any specific higher-level control abstraction that restricts usage of those entities. In order to facilitate the construction of higher-level abstractions without large numbers of opaque objects, EVPPath provides insight into such things as low-level resource availability by allowing Cod functions access to system state. Using these facilities, a higher-level control abstraction can, for example, use EVPPath to create an event delivery overlay that reconfigures itself in response to changes in computational demands so as to maintain a throughput goal. In this case, EVPPath could be used to both implement the data flows as well as the monitoring system that acquires resource data, filters it as necessary and forwards it to a central location for decisions on reconfiguration.

EVPPath’s type handling and response selection is designed to function efficiently in the common case of homogeneous event types or simple heterogeneity. It also functions effectively where there is less *a priori* knowledge of datatypes or when there might be multiple generations of clients operating simultaneously, as in an enterprise case, or where there are multiple valid interfaces to a software component, such as a numerical solver in scientific HPC. To support these scenarios, EVPPath allows for type flexibility whenever possible. For example, the input type specifications for EVPPath filter stones specify the minimum set of data fields that the function requires for operation. When presented with an event whose datatype is a superset of that minimum set, EVPPath will generate a customized version of the filter function that accepts the larger event type and conveys it unchanged if it passes the filter criterion. EVPPath type and function registration semantics provide sufficient information to allow dynamic morphing of message contents allowing new senders to seamlessly interact with prior-generation receivers as explored in earlier work [2].

EVPPath also combines some of this functionality in novel ways to meet the goals described in Section 2.1. For exam-

ple, EVPPath allows ‘congestion handlers’ to be associated with bridge stones. These functions are designed to be invoked when network congestion causes events to queue up at bridge stones rather than being immediately transmitted. Because EVPPath exposes network-level resource measures such as estimated bandwidth, and because EVPPath type semantics allow reflection and views into the application level data in events, congestion handlers have the ability to customize an event stream in an application-specified manner to cope with changing network bandwidth. This capability relies upon the run-time event filtering, open access to network resources, and insight into application-level types that are unique features of EVPPath. Because this feature is supported at the data transport level in EVPPath, it also allows higher-level control schemes to utilize this mechanism when and as appropriate.

4. APPLICATIONS AND PERFORMANCE

A full characterization of event system performance is complex, but it is interesting to examine how the building blocks described in the EVPPath architecture are actually used to implement event-based systems at various locations in the design space. This section examines the use of this architecture in application scenarios. In particular, we explore the use of EVPPath to:

- implement the event transport functionality in a different publish/subscribe system, ECho[10];
- implement peta-scalable I/O interfaces through in-network staging.

4.1 Implementing a full publish/subscribe system

The ECho publish/subscribe system is an efficient middleware designed for high-speed event propagation. Its basic abstraction is an event channel, which serves as a rendezvous point for publishers and subscribers and through which the extent of event notification propagation is controlled. Every event posted to a channel is delivered to all subscribers of that channel, in a manner similar to the CORBA Event Service [24].

ECho event channels, unlike many CORBA event implementations and other event services such as Elvin [25], are not centralized in any way. ECho channels are light-weight virtual entities. Figure 8 depicts a set of processes communi-

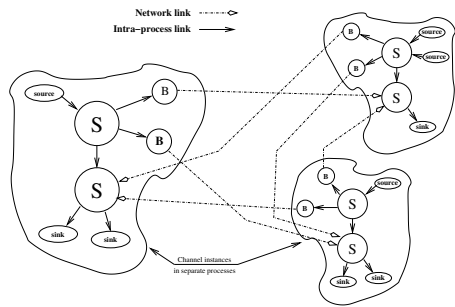


Figure 9: ECHO Event Channel implementation using EVPath stones.

cating using event channels. The event channels are shown as existing in the space between processes, but in practice they are distributed entities, with bookkeeping data residing in each process where they are referenced as depicted in Figure 8b. Channels are *created* once by some process, and *opened* anywhere else they are used. The process that creates the event channel is distinguished in that it is the contact point for other processes that use the channel.

However, event notification distribution is not centralized, and there are no distinguished processes during notification propagation. Messages are always sent directly from an event source to all sinks, and network traffic for individual channels is multiplexed over shared communications links. One of ECHO’s novel contributions is the concept of a *derived* event channel, a channel in which the events delivered to subscribers are a filtered or transformed version of the events submitted to another channel [10].

The style of event propagation provided by ECHO can be easily implemented through a stylized use of EVPath stones. Each open channel in a process is represented by two split stones as shown in Figure 9. The upper split stone is the stone to which events are submitted. Its outputs are directed to bridge stones that transmit the events to other processes where the channel is open and to the lower split stone. The lower split stone’s outputs are directed to terminal stones which deliver events to local subscribers on that process. While local events are always submitted to the upper split stone, the bridge stones that transmit events between processes target the lower split stone on remote hosts. This arrangement, maintained by ECHO’s subscription mechanism, duplicates the normal mechanics of ECHO event delivery.

From a control perspective, the EVPath re-implementation of ECHO leverages the pre-existing control messaging system that ECHO used. An initial enrollment message is sent from a new publisher and/or subscriber to the channel opener. In reply, the opener sends back a complete list of all of the publishers and subscribers and their contact information. In the EVPath implementation with a TCP transport, this contact include the IP address, port number, and lower stone ID number. The joiner then contacts each member on the list with an update to register itself as either a subscriber or publisher.

ECHO’s derived event channels can be implemented with a simple addition to the convention described above. When a derived event channel is created, ECHO creates an instance of the new channel on every process in which the original chan-

nel is represented. Then, the two-stone representation of the original channel and the derived channel are linked with a filter or transform stone as shown in Figure 10. This arrangement duplicates ECHO’s derived event channel semantics in which the filtering is always performed at the point of event submission in order to avoid unnecessary network overheads.

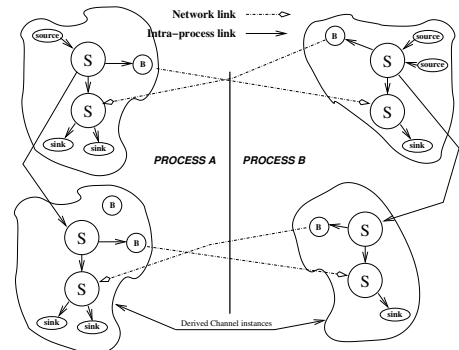


Figure 10: Derived Event Channel implementation using EVPath stones.

4.1.1 Filtering Efficiency

The earlier work on ECHO demonstrated the interesting result that, while abstractions like the derived event channel move more computation to the publisher of events, the total computational demands on the publisher can be lowered even if the filter actually rejects very few of the events it considers. This occurs because by discarding the events, the publisher can avoid the overhead of actually executing the TCP stack and transmitting the events. Figure 11 depicts the total CPU time incurred by a process sending 2 million relatively small (1000 byte) records as a function of the rejection ratio of the filter. The EVPath implementation of ECHO event distribution shows a similar effect, with the filter reducing CPU overheads with less than a 10% rejection ratio.

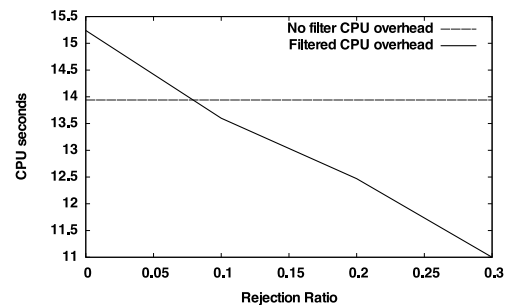


Figure 11: CPU overhead as a function of filter rejection ratio.

4.1.2 Network Operation

In order to evaluate EVPath’s network operation, we employ a simple stone configuration, submitting fully typed events directly to an output stone on process A where they will be transmitted to process B and delivered to a terminal

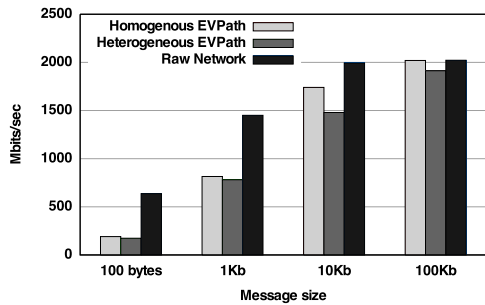


Figure 12: EVPPath throughput for various data sizes

stone. Figure 12 compares EVPPath’s delivered bandwidth to available bandwidth (as measured by netperf [17]) over a range of data sizes. Note that while netperf is just delivering a block of bytes, EVPPath is delivering fully typed messages. This simple case of a transfer from a 64-bit x86 machine to a similar box with a terminal stone that is expecting the same data type that is sent is the best case for EVPPath, and as Figure 12 shows, EVPPath delivers nearly all of available bandwidth at large message sizes.¹

As a more complex test, Figure 12 also presents results from experiments where the sender’s data structure does not perfectly match that required by the receiver. We simulate a byte-endian mismatch and a transfer between 32-bit and 64-bit machines by having the original senders instead transmit data that was pre-encoded on other architectures. Both scenarios require a more complex decoding transformation on the receiving side before the data can be delivered to the application. Figure 12 shows that the byteswap decoding operation, handled with custom-generated subroutines in EVPPath, has no impact on delivered bandwidth on smaller data sizes and causes less than a 5% degradation (as compared to a homogeneous transfer) at the worst case.

4.2 I/OGraph

As a final example in the scientific computing space, we consider an I/O library for scalable petascale applications in the I/OGraph project. The client-side interface registers data buffers as ready to write, and these get turned into formatted messages which are published through an EVPPath overlay for sorting, naming, and delivery to the process which commits the data to disk.

A key feature of this work is the idea of delaying global metadata consistency for scalable I/O – for example, the process of creating a particular file and getting a shared file pointer (or agreeing on how to partition into non-overlapping file regions for non-shared pointers) can be prohibitively expensive when scaled to tens of thousands of nodes or more. By enabling this metadata to be generated and associated with the data packet somewhere down stream in the network, the blocking time for the application can be reduced. For example, data can go through partial aggregation as it is gathered into the storage nodes, making for a much faster calculation of the prefix sums for writes into a shared file. Indeed, some application domains may prefer that the

¹These measurements here are obtained on a RedHat Linux cluster of dual socket, quad-core Intel-based blades, using non-blocking Infiniband networking. The nodes are provisioned with 1 Gigabyte of RAM per core; a typical if not low ratio for current high performance computing deployments.

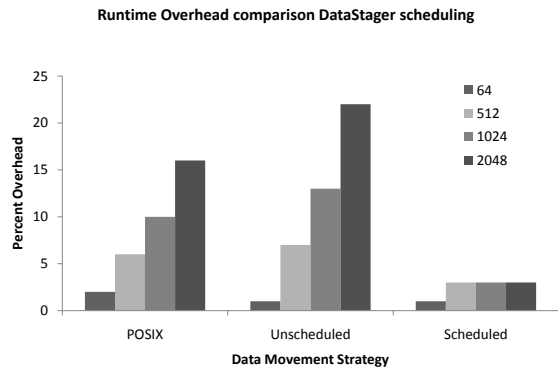


Figure 13: GTC run time increases due to event I/O with different approaches

data not be gathered into a shared file at all, if there is a mechanism for doing intelligent aggregation; for example, generating files based on the quadrant of simulation space, rather than the particular order of commits from the compute nodes.

Another key feature of this work is that exactly these sorts of domain-specific customizations can be accommodated using an imperative-style data treatment specification. Coupling the queue logging functionality of EVPPath that was discussed in the previous section with the arbitrary imperative data parsing abilities of Cod and with the router stone shown in Figure 4 leads to a new type of functionality. As messages arrive, they are enqueued and a custom function is invoked to evaluate them. However, that function has full control of its queue – it can publish events, delete events from the queue, create and publish synthetic events (*e.g.* windowed averages), and provide in-situ coordination.

The I/OGraph infrastructure can leverage this feature of the EVPPath data handling layer to implement the delayed metadata assignment in a clean way. Application-specific modules must only determine the thresholding criteria for when certain pieces of data should be written together; these could be physical (queue length ≥ 8) or domain driven (data outside of bounding box B). The I/OGraph library will create the EVPPath overlay and place the queue handling function on one or more nodes that are set aside as a data staging partition. Because the data is published in a single act from the queue handler, any relevant file data (such as prefix sums for the individual pieces) can be much more easily determined.

4.2.1 Data Extraction

The use of asynchronous event-style communication for data transfer can reduce or eliminate the blocking time experienced by HPC codes using synchronous methods for I/O, but a resulting new problem is one of potential perturbation in communication times experienced by the tightly coupled and finely-tuned parallel application. Consider the graph in Figure 13 which shows the increase in run-time due to asynchronous event export from a Gyrokinetic Turbulence Code [22] (GTC fusion modeling) running on the Oak Ridge National Laboratory’s Cray XT Jaguar machine. The values shown represent percent increased run-time as compared to no I/O scaling the number of processors in the parallel execution from 64 to 2048.

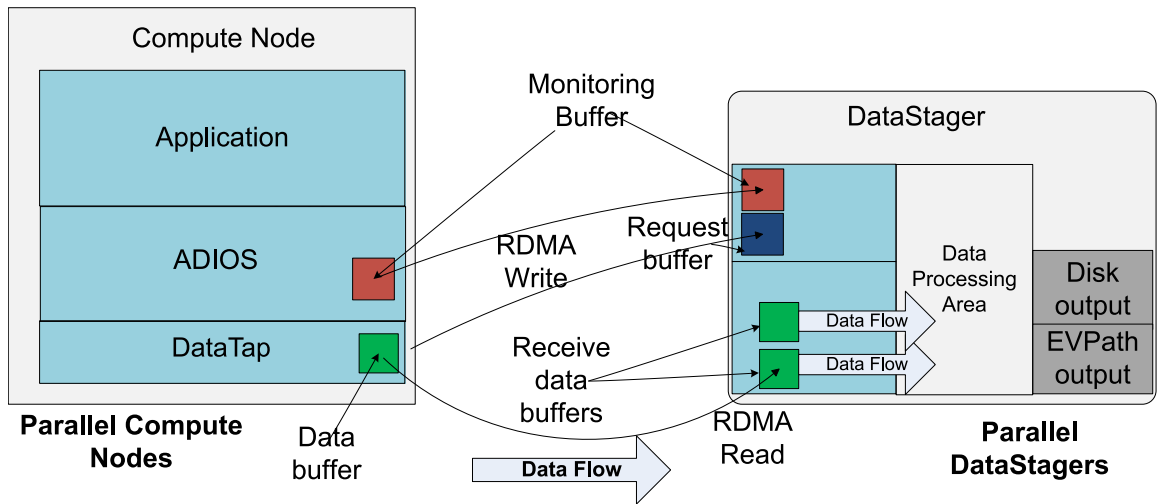


Figure 14: DataStager Architecture

In particular, consider the left-most and center groups of values which represent POSIX and ‘unscheduled’ asynchronous I/O. The POSIX approach simply involves an I/O infrastructure doing network write(s) when events are submitted in each compute node, while in the ‘unscheduled’ approach, RDMA requests are initiated by an I/O node to gather events asynchronously without regard to the compute state of the application. While acceptable for small numbers of processors, these naive approaches to event I/O suffer from competition for network resources and interference with the parallel application’s core MPI communication as the application is scaled to larger numbers of processors.

In order to address this problem and make the EVPath infrastructure useful for data extraction in this HPC environment, we developed the DataStager [1] shown in Figure 14. This architecture leverages the penetration of RDMA-based network infrastructures such as Infiniband, iWarp, Quadrics Elan, IBM BlueGene Interconnect, and Cray SeaStar in modern supercomputing infrastructures. Specifically, these provide a new opportunity for the creation of a staging service that shifts the burden of synchronization, aggregation, collective processing, and data validation to a subset of the compute partition – the staging area. Data processing in the staging area can then be used to provide input to a variety of extensible service pipelines such as those needed for formatted data storage on disk [21], for data movement to secondary or even remote machines [15], or for data visualization [30].

While data staging action may span the gamut of EVPath actions, we consider here only the I/O performance and overheads that result from its scheduling behaviour. In particular, we have designed four schedulers in order to evaluate their ability to enhance functionality, to provide improved performance, and to reduce perturbation for the application.

1. a constant drain scheduler,
2. a state-aware congestion avoidance scheduler,
3. an attribute-aware in-order scheduler, and
4. a rate limiting scheduler.

The details of those scheduling techniques are beyond the scope of this paper, but as the right-most column of Figure 13 shows, they allow significant asynchronous I/O to be achieved from the GTC application while keeping applica-

tion perturbation to around 4%.

5. SUMMARY

In this paper, we have briefly surveyed the design space for high performance event-based publish/subscribe systems and more deeply explored EVPath, an infrastructure aimed at supporting applications and middleware with high performance eventing needs. This is an area of continuing research and many topics are beyond the scope of a single paper. In particular, as multicore computing platforms continue to evolve, the importance of messaging in general and event-based approaches in particular would seem to be growing. With deep memory hierarchies and System-on-a-Chip architectures, messaging as a communication approach between “clumps” of cores would seem to be a viable and scalable approach. Additionally, the software trends towards composable, componentized applications are inevitable as well. So both from a platform perspective and from an application evolution perspective, event-based middleware will have a strong role to play in the future of high performance communications.

The work presented here, both generally and the authors’ specific research, is intended to spur further thought over the possible development of new techniques that might break down some of the classic design space trade-offs. Future high performance event messaging systems have the potential to become generically useful tools for high performance, alongside other middleware toolkits like MPI.

6. REFERENCES

- [1] H. Abbasi, M. Wolf, F. Zheng, G. Eisenhauer, S. Klasky, and K. Schwan. Scalable data staging services for petascale applications. In *hpd2009*, 2009.
- [2] S. Agarwala, G. Eisenhauer, and K. Schwan. Lightweight morphing support for evolving data exchanges in distributed applications. In *Proc. of the 25th International Conference on Distributed Computer Systems (ICDCS-25)*, June 2005.
- [3] K. Birman, A. Schiper, and P. Stephenson. Lightweight Causal and Atomic Group Multicast.

- ACM Transactions on Computer Systems*, 9(3), Aug. 1991.
- [4] D. Box, L. Cabrera, C. Critchley, F. Curbera, D. Ferguson, A. Geller, S. Graham, D. Hull, G. Kakivaya, A. Lewis, et al. Web Services Eventing (WS-Eventing). *W3C Member Submission*, 2006.
 - [5] L. Brenna, A. Demers, J. Gehrke, M. Hong, J. Ossher, B. Panda, M. Riedewald, M. Thatte, and W. White. Cayuga: a high-performance event processing engine. In *SIGMOD '07: Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, pages 1100–1102, New York, NY, USA, 2007. ACM.
 - [6] Z. Cai, G. Eisenhauer, Q. He, V. Kumar, K. Schwan, and M. Wolf. Iq-services: Network-aware middleware for interactive large-data applications. *Concurrency & Computation. Practice and Experience Journal*, 2005.
 - [7] A. Carzaniga, D. S. Rosenblum, and A. L. Wolf. Challenges for distributed event services: Scalability vs. expressiveness. In *Proc. of Engineering Distributed Objects (EDO '99), ICSE 99 Workshop*, May 1999.
 - [8] A. Demers, J. Gehrke, M. Hong, M. Riedewald, and W. White. A general algebra and implementation for monitoring event streams. Technical Report TR2005-1997, Cornell University, 2005.
 - [9] A. Demers, J. Gehrke, M. Hong, M. Riedewald, and W. White. Towards expressive publish/subscribe systems. *Lecture Notes in Computer Science*, 3896:627, 2006.
 - [10] G. Eisenhauer, F. Bustamante, and K. Schwan. Publish-subscribe for high-performance computing. *IEEE Internet Computing - Asynchronous Middleware and Services*, 10(1):8–25, January 2006.
 - [11] G. Eisenhauer and L. K. Daley. Fast heterogeneous binary data interchange. In *Proc. of the Heterogeneous Computing Workshop (HCW2000)*, May 3-5 2000.
 - [12] P. Eugster. Type-based publish/subscribe: Concepts and experiences. *ACM Trans. Program. Lang. Syst.*, 29(1):6, 2007.
 - [13] P. Eugster, P. Felber, R. Guerraoui, and A.-M. Kermarec. The many faces of publish/subscribe. Tech. Report DSC-ID:200104, École Polytechnique Fédérale de Lausanne, Lausanne, France, January 2001.
 - [14] I. Foster, K. Czajkowski, D. Ferguson, J. Frey, S. Graham, T. Maguire, D. Snelling, and S. Tuecke. Modeling and managing state in distributed systems: The role of OGSF and WSRF. *Proceedings of the IEEE*, 93(3):604–612, 2005.
 - [15] M. K. Gardner, W.-c. Feng, J. S. Archuleta, H. Lin, and X. Ma. Parallel Genomic Sequence-Searching on an Ad-Hoc Grid: Experiences, Lessons Learned, and Implications. In *ACM/IEEE SC'06: The International Conference on High-Performance Computing, Networking, Storage, and Analysis*, Tampa, FL, November 2006. Best Paper Nominee.
 - [16] A. Grimshaw, M. Morgan, D. Merrill, H. Kishimoto, A. Savva, D. Snelling, C. Smith, and D. Berry. An open grid services architecture primer. *Computer*, 42(2):27–34, 2009.
 - [17] Hewlett-Packard. The netperf network performance benchmark. <http://www.netperf.org>.
 - [18] Y. Huang, A. Slominski, C. Herath, and D. Gannon. Ws-messenger: A web services based messaging system for service-oriented grid computing. In *6th IEEE International Symposium on Cluster Computing and the Grid (CCGrid06)*, 2006.
 - [19] N. Jiang, A. Quiroz, C. Schmidt, and M. Parashar. Meteor: a middleware infrastructure for content-based decoupled interactions in pervasive grid environments. *Concurr. Comput. : Pract. Exper.*, 20(12):1455–1484, 2008.
 - [20] V. Kumar, B. F. Cooper, Z. Cai, G. Eisenhauer, and K. Schwan. Resource-aware distributed stream management using dynamic overlays. In *Proceedings of the 25th IEEE International Conference on Distributed Computing Systems (ICDCS-2005)*, 2005.
 - [21] J. Lofstead, K. Schwan, S. Klasky, N. Podhorszki, and C. Jin. Flexible io and integration for scientific codes through the adaptable io system (adios). In *Challenges of Large Applications in Distributed Environments (CLADE)*, 2008.
 - [22] L. Oliker, J. Carter, michael Wehner, A. Canning, S. Ethier, A. Mirin, G. Bala, D. parks, patrick Worley Shigemune Kitawaki, and Y. Tsuda. Leading computational methods on scalar and vector hpc platforms. In *Proceedings of SuperComputing 2005*, 2005.
 - [23] O. M. G. (OMG). Notification service specification 1.0. <ftp://www.omg.org/pub/doc/formal/00-06-20.pdf>, June 2000.
 - [24] O. M. G. (OMG). Event service specification 1.1. <ftp://www.omg.org/pub/docs/formal/01-03-01.pdf>, March 2001.
 - [25] B. Segall and D. Arnold. Elvin has left the building: A publish/subscribe notification service with quenching. In *Proc. of the AUUG (Australian users group for Unix and Open Systems) 1997 Conference*, September 1997.
 - [26] D. Skeen. The enterprise-capable publish-subscribe server. <http://www.vitria.com>.
 - [27] R. Strom, G. Banavar, T. Chandra, M. Kaplan, K. Miller, B. Mukherjee, D. Sturman, and M. Ward. Gryphon: An information flow based approach to message brokering. In *International Symposium on Software Reliability Engineering '98 Fast Abstract*, 1998.
 - [28] Tibco. TIB/rendezvous. <http://www.rv.tibco.com/rvwhitepaper.html>.
 - [29] M. Wolf, H. Abbasi, B. Collins, D. Spain, and K. Schwan. Service Augmentation for High End Interactive Data Services. In *Proceedings of Cluster 2005*, 2005.
 - [30] M. Wolf, Z. Cai, W. Huang, and K. Schwan. Smartpointers: Personalized scientific data portals in your hand. In *Proceedings of the Proceedings of the IEEE/ACM SC2002 Conference*, page 20. IEEE Computer Society, 2002.