ECho Event Delivery Middleware

Greg Eisenhauer
College of Computing
Georgia Institute of Technology

The general paradigm of event-based programming has received added attention as the growth of the Internet has led to the creation of more complex enterprise-wide distributed systems. In event-based programming, event suppliers generally are unaware of the number or location of any consumers for the events they produce. Relying on event mechanisms in the design and composition of distributed systems tends to produce systems which are less tightly-coupled. Event-based communication mechanisms are receiving wide use in:

- systems involving mobility, both in mobile code (agent) systems and in support of mobile computers;
- cooperative systems that support collaboration between multiple users, from simple shared whiteboards to distributed virtual reality;
- adaptive systems which react to changes in resources and demands to maintain a quality of service;
- extensible systems where events serve as a 'glue' connecting components; and
- all types of monitoring applications, from telecommunications network monitoring to application-level monitoring of running scientific simulations.

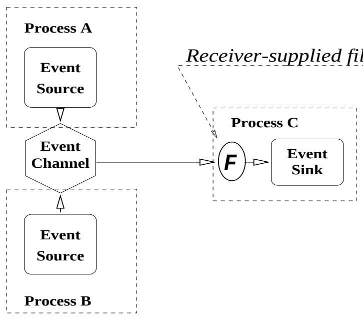
In current practice, these applications are supported by event systems with vastly different features and performance characteristics, ranging from statically-typed single-address-space implementations operating at rates of millions of events per second to distributed object-based event channels delivering throughput in the range of tens to hundreds of events per second. The dramatically lower event rates of distributed object system implementations are partly attributable to network delays, but to a larger extent they are caused by the overheads imposed by full object systems with features such as reflection (through which programs can discover the methods and attributes of event objects without a-priori knowledge). While the slower speeds are a distinct disadvantage, subclassing and reflection can be essential features in creating robust systems. In particular, reflection allows the creation of generic event-processing components that can be dropped into event streams. Subclassing allows event information exchange to evolve as a system evolves without requiring the simultaneous update of all components. Both of these features are invaluable in dealing with the application integration issues encountered in deploying and updating large distributed systems.

In addition to the advantages offered by object systems in general, event systems based on Java have potential to offer event filtering and data reduction through the interposition of third-party objects. When interposition is combined with code mobility, it creates potential for receivers to transparently customize event traffic to precisely match their needs, performing filtering or data reduction at the event source and avoiding network overheads. This can be a significant performance optimization for some applications, potentially reducing their network usage by orders of magnitude.

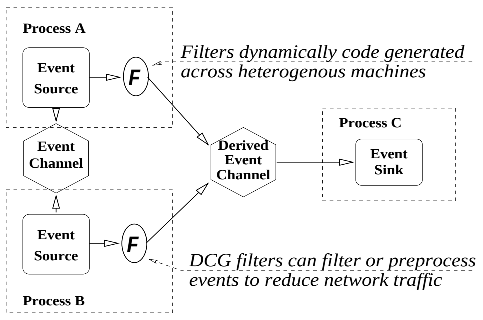
ECho is a distributed event delivery system that combines the best aspects of prior work. In particular, our goal was to create an event system that:

- provides high-performance shared-memory communication,
- offers performance similar to traditional message passing for data-intensive distributed applications,
- provides application-integration advantages similar to those offered by object systems, and
- implements source-side data filtering and data reduction efficiently enough to benefit communication-intensive applications.

One of the differentiating characteristics of ECho is its support for efficient transmission and handling of fully typed events. Some event delivery systems leave event data marshalling to the application. ECho allows types to be associated with event channels, sinks and sources and will automatically handle heterogeneous data transfer issues. Building this functionality into the ECho using PBIO allows for efficient layering that nearly eliminates data copies during marshalling and unmarshalling. Careful layering to minimize data



(a) Filtering an event stream


Process A
Event Source
F
Filters dynamically code generated across heterogenous machines
Event Channel
Derived Event Channel
Process C
Event Sink
Event Source
F
DCG filters can filter or preprocess events to reduce network traffic
Process B

(b) Derived channel with filter migration

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.

A second distinguishing characteristic of ECho is that it supports the robust evolution of sets of programs communicating with events by allowing variation in data types associated with a single channel. In particular, ECho allows an event source to 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 feature can be extremely 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 intraprocess 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.

In addition to high efficiency in local and network event delivery, ECho makes a significant contribution to the state of the art in that it allows event receivers to easily and efficiently customize the nature and content of the event traffic that is sent to them. Many event-based application discard large numbers of unwanted events because of a lack of a good mechanism for suppressing them. These useless events take up network resources and consume CPU time for the receiver to process, filter and discard them, as shown in Figure (a) above. ECho's approach to this involves extending event channels with the concept of a *derived* event channel, depicted in Figure (b). Rather than requiring the receiver to filter incoming events, we create a new event 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 event that results in the derived channel. This approach has the advantage that we limit unwanted event traffic (and the associated waste of compute and network resources) as much as possible.

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 sink but must be evaluated at the (possibly remote) source, a simple function pointer is obviously insufficient. 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 impacts efficiency and requires a potentially large interpreter environment everywhere event channels are used. The approach taken in ECho preserves both expressiveness and efficiency. The function F is expressed in E-Code, a limited subset of C, and dynamic code generation is used to create a native version of F on the source host. ECho's dynamic code generation makes source-side filtering so efficient that filters can execute in less time than it take to transmit even a small event. So, while filtering in general expends additional source-side computing cycles to save network bandwidth, ECho's efficiency often results in clear benefits for *all* parties in the communication.

ECho runs on a variety of platforms including Sun Sparc Solaris 2.x (32 and 64-bit), Sun Sparc SunOS 4.1.3, SGI MIPS IRIX 5.x, SGI MIPS IRIX 6.x (32 and 64-bit), IBM RS6000 AIX 3.2, x86 Linux, x86 Solaris 2.x, and x86 Windows NT.

Additional information, documentation and source for ECho can be retrieved from: `http://www.cc.gatech.edu/systems/projects/ECho/`.