The ECho Event Delivery System

Greg Eisenhauer
eisen@cc.gatech.edu

College of Computing
Georgia Institute of Technology
Atlanta, Georgia 30332

July 18, 2002 – ECho Version 2.1

1 Introduction

ECho is an event delivery middleware system developed at Georgia Tech. Superficially, the semantics and organization of structures in ECho are similar to the Event Channels described by the CORBA Event Services specification[COSEvents]. Like most event systems, what ECho implements can be viewed as an anonymous group communication mechanism. In contrast with more familiar one-to-one send-receive communication models, data senders in anonymous group communication are unaware of the number or identity of data receivers. Instead, message sends are delivered to receivers according to the rules of the communication mechanism. In this case, event channels provide the mechanism for matching senders and receivers. Messages (or *events*) are sent via *sources* into *channels* which may have zero or more *subscribers* (or *sinks*). The locations of the sinks, which may be on the same machine or process as the sender, or anywhere else in the network, are immaterial to the sender. A program or system may create or use multiple event channels, and each subscriber receives only the messages sent to the channel to which it is subscribed. The network traffic for multiple channels is multiplexed over shared communications links, and channels themselves impose relatively low overhead. Instead of doing explicit *read()* operations, sink subscribers specify a subroutine (or handler) to be run whenever a message arrives. In this sense, event delivery is asynchronous and passive for the application.

Figure 1 depicts a set of processes communicating with event channels. The event channels are shown as existing in the space between processes, but in practice they are distributed entities, with bookkeeping data in each process where they are referenced. Channels are *created* once by some process, and *opened* anywhere else they are used. The process which creates 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 contact information for the creating process as well as information identifying the specific channel. However, event distribution is not centralized and there are no distinguished processes during event propagation. Event messages are always sent directly from an event source to all subscribers. Because of this setup, if the process which created an event channel exits the channel can no longer be opened by other processes. However, the channel will continue to function for event distribution between the process that have already opened it.

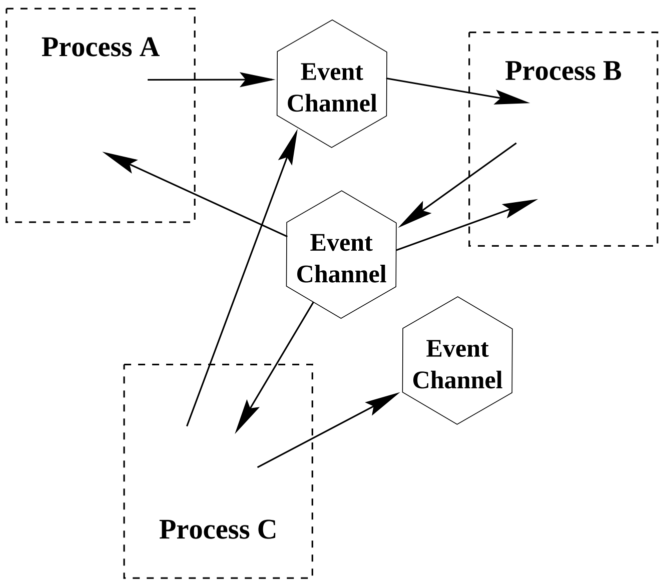

Figure 1: Processes using Event Channels for Communication.

2 ECho API

ECho is largely built on top of the Connection Manager (CM)[Eisen00CM]. The next sections provide the basic ECho API and discuss their implementation.

2.1 ECho Control Contexts

Many ECho entities and operations are associated with a "control context" or `EControlContext` variable. These variables are the mechanism through which threads of control are associated with synchronous and asynchronous actions in ECho. Later sections will expand up the use of `EControlContext` variables, but many ECho programs have only a single context variable associated with an instance of the Connection Manager. In particular, the `ECho_CM_init()` function initializes a Connection Manager for ECho activity and returns an `EControlContext` variable which can be used in subsequent ECho calls. This initialization allows ECho to piggyback on the Connection Manager and use the thread of control which is used by CM for servicing the network.

In particular, there are a couple of common ways to structure the control flow in ECho/CM programs. In the simplest, non-threaded program the control flow looks like this:

```
int main()
{
    CManager cm;
    EControlContext ec;

    /*  ...  */
    cm = CManager_create();
    ec = ECho_CM_init(cm);
    /*  ...  */
    CMrun_network(cm);  /* handle network events in in CM/ECho */
}
```

In this program, the `CMrun_network()` is a blocks forever (or until the CM is shut down, servicing any network connections established by CM.

For threaded programs, CM relies on the **gen_threads** package to provide a generic wrapper around threads libraries. After **gen_threads** has been initialized by calling a specific init function, the subroutine `CMfork_comm_thread()` causes CM to fork a kernel-level thread to handle network traffic. ECho will also use this thread for various asynchronous tasks. A sample control structure for a threaded program using Pthreads is given below:

```
int main()
{
    CManager cm;
    EControlContext ec;
    int forked;

    gen_pthread_init();

    cm = CManager_create();
    forked = CMfork_comm_thread(cm);
    if (!forked) {
        printf("Fork of communications thread failed, exiting\n");
        exit(1);
    }
    ec = ECho_CM_init(cm);

    /*
     * communications thread is forked - main() should not return
     * until time for program exit.
    */
}
```

A variety of other control schemes are discussed in the CM documentation, Section 4.2. This documentation is available at `http://www.cc.gatech.edu/systems/projects/CM/`.

2.2 Channel Creation and Subscription

ECho Event Channels are created with the `EChannel_create()` call given below:

```
EChannel EChannel_create(EControlContext ec);
```

Channel creation essentially involves initializing a data structure that will record information about the subscribers to the event channel. The process in which the creation is performed is the permanent contact point for the event channel and participates in the subscription and unsubscription process. However, it does not necessarily participate in event propagation.

When they are created, channels are assigned a character-string global ID. This string can be used to open the channel for use in other processes. The calls involved are given below:

```
char *ECglobal_id(EChannel chan);
EChannel EChannel_open(EControlContext ec, char *global_id);
```

Opening a channel that was created in another process involves obtaining a CM connection to the creating process and obtaining from that process a list of other processes which have opened the channel. CM connections are then obtained to each process in this list. These direct connections will be used for event delivery.

2.3 Event Submission

Several subroutines make up the event submission API in the ECho. The first of these is `ECsource_subscribe()`, which derives an `ECSourceHandle` to which an event can be submitted. Given an `ECSourceHandle`, events can be submitted to the channel with `ECsubmit_event()` given below:

```
ECSourceHandle ECsource_subscribe ( EChannel chan );
void ECsubmit_event ( ECSourceHandle handle, void * event, int event_length );
```

Event submission is synchronous in the sense that before the submission call returns, any local handlers for the event have been called, the event has been transmitted to any remote sinks and the application is free to destroy the data that was submitted.

2.4 Event Delivery

Events are delivered to event sinks that have registered handlers. Handler registration is accomplished through the `EChannel_subscribe_handler()` function:

```
typedef void (*ECHandlerFunction) (void *event, int length, void *client_data);
ECSinkHandle ECsink_subscribe( EChannel chan, ECHandlerFunction func, void *client_data);
```

When an event arrives from the network or is submitted to a channel that has a local sink, the event is queued for later delivery. Event delivery consists of calling the specified handler function and passing the event as a parameter. The `client_data` value specified in the subscribe call is also passed to the handler without interpretation by the library.

Event delivery is done at two points, just before `ECsubmit_event()` returns and at the end of network event handling in CM. In each case, all pending events are dispatched. During dispatch, handler functions are called directly. Therefore, handler functions that can potentially block or compute for long periods of time should fork a thread to perform those tasks. Preservation of the event parameter data is the responsibility of the handler.

2.5 Simple Example Programs

Figures 2, 3 and 4 give simple independent programs for creating an event channel, submitting events to a channel and establishing a handler for events in a channel. The programs are similar in that they all involve a setup phase where they create a CM and initialize ECho on it to create a EControlContext, and a phase where they handle network traffic (via some variant of `CMrun_network()`, `CMpoll_network()`, `CMsleep()`, etc.). The channel creation program in Figure 2 is the simplest. Between the CM setup and event handling phases, all it does is create an event channel with `EChannel_create()` and print out it's ID as given by `ECglobal_id()`. This ID is used by the event source and sink programs to access the channel. This example sends a long integer which is initialized based on the time and incremented with each event submission.

```
int main()
{   /* this program creates an event channel */
    EChannel chan;
    CManager cm;
    EControlContext cc;

    cm = CManager_create();
    cc = ECho_CM_init(cm);

    chan = EChannel_create(cc);
    printf("Channel ID is: %s\n", ECglobal_id(chan));
    CMsleep(cm, 600);   /* handle net for 10 min */
    return 1;
}
```

Figure 2: Simple channel creation program.

The program for sending events sending (source program) is slightly more complex than the creation program. It creates a CM and initializes ECho to create an EControlContext, it uses `EChannel_open()` to connect to the channel created by the program in Fig. 2. The second argument to `EChannel_open()` is the string channel ID that is printed out when the channel creation runs.[1] Once the channel is opened, the program identifies itself as an event source for that channel (and gets a source handle) by calling `ECsource_subscribe()`. After this, the sample program enters a loop where it submits events and uses `CMsleep()` to handle the network and delay briefly between event submissions. The three arguments to `ECsubmit_event()` are the `ECsource_handle` from the `ECsource_subscribe()` call and the base address and length of the event data to send. The event system does not interpret the event data, only delivers it to any sink subscribers.

```
int main(argc, argv)
int argc;
char **argv;
{
    CManager cm;
    EChannel chan;
    EControlContext cc;
    ECSourceHandle handle;
    long a = time(NULL) % 100;

    cm = CManager_create();
    cc = ECho_CM_init(cm);

    chan = EChannel_open(cc, argv[1]);
    handle = ECsource_subscribe(chan);
    while (1) {
        printf("I'm submitting %ld\n", ++a);
        ECsubmit_event(handle, &a, sizeof(a));
        CMsleep(cm, 5);
    }
}
```

Figure 3: Simple event source program.

Figure 4 shows a simple program which receives events. Like the event source program in Figure 3, the sink program uses `EChannel_open()` to open the channel who's ID is passed in on the command line. After opening the channel, the program uses `ECsink_subscribe()` to register a subroutine to handle events in that channel. Finally calls CMsleep() to handle network messages for two minutes. When an event arrives, the event handler subroutine is called with the base address of the event data, the length of the data, and the `client_data` value which was specified as the third parameter to `ECsink_subscribe`. In this case, the `client_data` parameter is unused, but it could give application-specific information to the handler function.

```
void handler(event, length, client_data, attrs)
void *event;
int length;
void *client_data;
attr_list attrs;
{
    printf("event data is %ld\n", *(long *) event);
}

int main(argc, argv)
int argc;
char **argv;
{
    CManager cm;
    EChannel chan;
    EControlContext cc;

    cm = CManager_create();
    cc = ECho_CM_init(cm);

    chan = EChannel_open(cc, argv[1]);
    (void) ECsink_subscribe(chan, handler, NULL);
    CMsleep(cm, 120);
    return 0;
}
```

Figure 4: Simple event handling program.

The programs given in this section are very simple examples of channel creation,

[1]The ECho API doesn't provide any explicit mechanisms for communicating event channel IDs between programs. These simple example programs require the user to pass them in as command line arguments. Other programs might write them to files, send them in CM messages, or exchange them in some other way. Also, note that the contact information for ECho version 2 event channels contains shell meta-characters. These characters must be quoted to avoid interpretation by the shell.

submission and handling programs, however they should be sufficient to demonstrate the basic principles of the channels. Note that you can run any number of event source or sink programs[2] and that each sink will receive any event submitted by any source. Also notice that the channel provides no buffering. That is, events are distributed to the sinks that exist at the time the event is submitted by a source. Late joining sinks may miss early events. It's also worth noting that while these programs show event functionality divided into separate programs, that is only done for simplicity of presentation. In reality a single program can create any number of channels, and submit events to those or other channels and register handlers for any channel.

3 Sending Events in a Heterogeneous Environment

One glaring deficiency in the example programs in the previous section is that they do not work correctly in a heterogeneous environment where the representation of "long" may differ from machine to machine. The event channel system itself will work because it is based upon CM mechanisms which handle representation differences. However, because the event channels treat event data as a simple block of bytes, binary event data (like the "long" value in the example) created on one machine may not be interpretable on another machine.

There are several possible approaches to this problem, including using `sprintf()/sscanf()` to encode/decode the data in ascii, or using htonl/ntohl to convert binary data to/from a standard byte order for transmission. Another option in this environment is to use PBIO directly or to use the *user format* facilities of CM. CM user formats are a layer on top of PBIO that can utilize CM for format transmission and avoid the use of a separate PBIO format server. This makes the service more useful in a variety of environments where socket connections to a PBIO format server are impossible or undesirable (such as from within the OS kernel). However, those routines are complex and their proper use is difficult. Therefore in ECho version 2, we recommend the use of **typed event channels**, described below.

3.1 Typed Event Channels

Typed event channels provide most of the functionality that is possible with CM user formats or direct PBIO, but with considerably reduced complexity. However, the use of typed event channels imposes some restrictions. For example, if PBIO is used directly, programmers have more flexibility in terms of submitting records of multiple types in the same event channel and using customized decoding and handling for each type. The typed event channel interface restricts event channels to carrying a single event type, but for that event type it handles all the conversions necessary to exchange binary data across heterogeneous machines. The remainder of this section details their use and interface.

Unlike untyped event channels, typed channels require type information to be specified at the time of channel creation. PBIO field lists are used to specify this information and the list of field names and types become the "type" of the event channel. (Field size and offset can be also be specified in the IOFieldList, but those values do not become part of the channel type specification.) In order to allow specifications of the field lists associated with structured subfields in the type, a

[2]While the number of event sources and sinks is theoretically unlimited, there are practical limits. In particular, since event distribution is not centralized, each source process creates a CM connection to each sink process. Most operating systems establish limits as to the number of simultaneous sockets or file descriptors that may be in use at one time. Programs using event channels may run into this limit when a large number of sinks are present.

null-terminated list of subfield type names and field lists can also be specified in the `subformat_list` parameter.

```
typedef struct _CMformat_list {
    char *format_name;
    IOFieldList field_list;
} CMFormatRec, *CMFormatList;

EChannel EChannel_typed_create(EControlContext ec, IOFieldList field_list,
                               CMFormatList subformat_list);
```

If there are no structured subfields, the `subformat_list` parameter should be NULL. The code to create a typed event channel which carries a record with a nested subformat looks like this:

```
typedef struct R3vector_struct {
    double x, y, z;
} R3vector;

typedef struct particle_struct {
    R3vector   loc, deriv1, deriv2;
} particle;

static CMFormatRec particle_format_list[] = {
    {"R3vector", R3field_list},
    {NULL, NULL},
};

static IOField R3field_list[] = {
    {"x", "float", sizeof(double), IOOffset(R3vector*, x)},
    {"y", "float", sizeof(double), IOOffset(R3vector*, y)},
    {"z", "float", sizeof(double), IOOffset(R3vector*, z)},
    {NULL, NULL, 0, 0},
};

static IOField particle_field_list[] = {
    {"loc", "R3vector", sizeof(R3vector), IOOffset(particle*, loc)},
    {"deriv1", "R3vector", sizeof(R3vector), IOOffset(particle*, deriv1)},
    {"deriv2", "R3vector", sizeof(R3vector), IOOffset(particle*, deriv2)},
    {NULL, NULL, 0, 0},
};
chan = EChannel_typed_create(ec, particle_field_list, particle_format_list);
```

Please note that both the field list and the format list are declared static. These lists must remain valid for the duration of the channel's existence. Most commonly they will be known statically at compile time and can be declared as in this example. They should *not* be stack-allocated variables. They can be dynamically allocated, but in that case the programmer is responsible for freeing those structures after the channel is destroyed.

Once a typed channel is created remote access can be accomplished through the standard `ECglobal_id()` and `EChannel_open()` calls. However, there are special event submission and sink and source subscribe calls as given below:

```
ECSourceHandle ECsource_typed_subscribe (EChannel chan, IOFieldList field_list,
                                         CMFormatList format_list);
void ECsubmit_typed_event (ECSourceHandle handle, void *event);
```

```
typedef void (*ECTypedHandlerFunction) (void *event, void *client_data);
ECSinkHandle ECsink_typed_subscribe (EChannel chan, IOFieldList field_list,
                                     CMFormatList format_list,
                                     ECTypedHandlerFunction func, void *client_data);
```

Essentially, these calls are the same as the corresponding untyped calls except for the addition of the `field_list` and `format_list` parameters to the subscribe functions and the elimination of the event length parameter in the submit and handler functions. The `field_list` and `format_list` specified in the subscribe calls *must* be appropriate for the type of channel they are applied to. Generally that means that the field names and types specified in the subscribe `field_list` and `format_list` must exactly match those in the `EChannel_typed_create()` call.[3] Note that while the field sizes and offsets in each of these calls must exactly specify the formats of those records on the machine executing the call, there is no requirement that these match those specified for the channel or any other sink. Any necessary transformations are provided by the channel. Additionally, if a channel is created with `EChannel_typed_create()`, all source and sink subscriptions must be typed.

4 Events and Threads

The previous examples in this document have all assumed non-threaded execution environment. In moving to threaded environments there are a number of issues to clarify, including questions of simultaneous use of event channels, determining which thread actually executes the event handlers and thread safe use of PBIO. This section will attempt to address these issues.

4.1 Thread basics

All CM routines, including the ECho routines, are capable of performing locking to protect their own data structures from simultaneous access. To avoid having different versions of CM for each thread package which a program might use, CM uses a package called gen_threads. Gen_threads is a generic wrapper for threads packages which provides an abstract thread interface implemented with function pointers. Gen_thread initialization calls fill in function pointers appropriate to specific thread packages so that CM routines can use them. If you use CM in a threaded environment, you must initialize gen_threads before calling `CManager_create()` so that the appropriate locks are created. If you initialize gen_threads later something bad is likely to happen.

Currently there are three different sets of wrappers with gen_threads. `gen_pthread_init()` will initialize gen_threads to use the Pthreads package. Calling `gen_cthread_init()` sets up the Georgia Tech Cthreads package instead. In addition to linking with the genthreads library these routines also require you to link with the appropriate external libraries to support the threads package. Initializing gen_threads appropriately is necessary to protect CM, but you also have the option of using the gen_threads interface in your application. Doing this instead of directly calling the threads package of your choice may make your application more portable in the sense that you may not have to rewrite your code to change threads packages. However, be warned that

[3]There is some room for variance here. In particular, it is ok to provide *more* information than necessary and to ask for *less* information than is available. So the fields and formats provided for the source subscribe can be a superset of those associated with the channel. Conversely the fields and formats provided for the sink subscribe can be a subset of those of the channel. Excess information is discarded at the sink end, so it does travel over the network.

```
int    gen_thr_initialized();
thr_thread_t thr_fork(void_arg_func func, void *arg);
void   thr_thread_detach(thr_thread_t thread);
void   thr_thread_yield();
thr_mutex_t thr_mutex_alloc();
void   thr_mutex_free(thr_mutex_t m);
void   thr_mutex_lock(thr_mutex_t m);
void   thr_mutex_unlock(thr_mutex_t m);
thr_condition_t thr_condition_alloc();
void   thr_condition_free(thr_condition_t m);
void   thr_condition_wait(thr_condition_t c, thr_mutex_t m);
void   thr_condition_signal(thr_condition_t c);
void   thr_condition_broadcast(thr_condition_t c);
void   thr_thread_exit(void *status);
int    thr_thread_join(thr_thread_t t, void **status_p);
thr_thread_t thr_thread_self();
```

Figure 5: Gen_threads API

gen_threads does not make all threads packages behave the same. For example, user-level and kernel-level threads packages behave differently when a thread does a blocking call. In the former case, all the threads in the process are blocked and in the latter only the calling thread is blocked. Using gen_threads does not hide this or other important semantic differences that may exist between thread libraries. It only provides a generic API to common thread functionality. The header file in Figure 5 summarizes the interface offered by gen_threads. The names of these calls link them with their obvious counterparts in common thread packages and should be familiar to most thread programmers. The only one which might be non-obvious is `gen_thr_initialized()` which simply returns true if gen_threads has been initialized.

Given that gen_thread is initialized, ECho programs can use CM's `CMfork_comm_thread()` subroutine to fork a thread that will devote itself to handling the network. This routine can be called anytime after creating the Connection Manager, but before any calls to `CMsleep()`, `CMpoll_network()`, etc.

4.2 Threads and Event Handler Routines

Multithreaded programs which use CM or ECho almost always devote a single thread to handling the network, often via `CMfork_comm_thread()`. A less common alternative which works out essentially the same is to have a single thread poll the network at intervals with `CMpoll_network()`. [4]

Given that there may also be other application threads which may submit events, create channels and/or register event handlers, which of these threads actually runs the event handlers? The answer depends upon where the event is coming from. If the event was submitted in another process and therefore arrives over the network, then the thread which is handling the network performs the call to the handler. One side effect of this is that the network will not be serviced during the time that an event handler is running. Because of this, long running handlers may want to fork a thread to complete their work. In the case of a handler forking a thread, be aware that the event data is only valid until the handler returns. If the forked thread needs access to the event data it must preserve it somehow.

[4] Any situation in which multiple threads call network handling functions `CMrun_network()` or `CMpoll_network()` is likely to have unpredictable results. Don't try this at home.

If a channel has event sources and sinks in the same process, then a different situation results and it is the thread which calls `ECsubmit_event()` which is borrowed to run the event handler. This situation has two side effects. First, as described in the paragraph above, long running handlers may want to fork a thread instead of delaying the submitting thread. Second, it is possible for handlers which themselves submit events to create an "event avalanche." That is, the first call to `ECsubmit_event()` results in a call to a local handler which itself calls `ECsubmit_event()` which calls a local handler which calls `ECsubmit_event()` which calls a local handler which calls `ECsubmit_event()`... This is not *per se* a problem for the event channel system as long as the chain ends eventually (otherwise it's an infinite recursion), but there may be other consequences with respect to stack usage and delay of the original submitting thread that may be of concern.

4.3 Assigning Handlers to a Specific Thread

ECho does offer a mechanism for overriding the normal way that event handlers are executed and instead assign them to a specific thread. More specifically, there is a channel subscribe function, `EChannel_subscribe_context()`, that allows event handlers to be assigned to a specific `EControlContext` value. Further, there is a function, `EContext_create()`, that creates `EControlContext` values that are not associated with `CManager` values and do not service the network. Specific threads can service these `EControlContext` values with the routines `EControl_poll` and `EControl_run`. These routines are roughly analagous to `CMpoll_network()` and `CMrun_network` respecively. `EControl_poll()` dispatches any events currently waiting on an `EControlContext` value and then returns. `EControl_run()` handles events continuously and does not return. The thread which calls one of these functions on the `EControlContext` which was used in a call to `EChannel_subscribe_context()` will then execute the handler.

The existence of this functionality complicates event submission somewhat. Previously, the callers of `ECsubmit_event` could assume that they were free to destroy event data as soon as that routine returned. This was a side effect of local events handlers being executed by the thread calling `ECsubmit_event`. However, if event handlers can be assigned to other threads, that assumption is no longer valid and a mechanism must be established for preserving event data until the handler has an opportunity to run. That mechanism is a new event submission routine, `ECsubmit_general_event()`. This routine has an additional parameter to allow the application to specify an appropriate deallocation routine for the event data. This allows the event system to keep the event data until all local handers have run and then deallocate it. The application should not reference the event data after it has been passed to `ECsubmit_general_event()`. *If* `ECchannel_subscribe_context()` *is used locally with a channel, then all event submissions to that channel must be done using* `ECsubmit_general_event()` *instead of* `ECsubmit_event()`. Since this applies only to local handlers, there is no global requirement to know if other processes use `ECchannel_subscribe_context()`, you only have to know if your program uses it.

To complement the `EChannel_subscribe_context()` and `ECsubmit_event()` calls described above, there are similar routines for typed event channels. The API for all these calls follows:

```
EControlContext EContext_create();
void EControl_poll (EControlContext);
oid EControl_run (EControlContext);

ECSinkHandle
ECsink_subscribe_context (EChannel chan, ECHandlerFunction func, void *client_data,
                          EControlContext ec);
```

```
ECSinkHandle
ECsink_typed_subscribe_context (EChannel chan, IOFieldList field_list, CMFormatList format_list,
                                ECTypedHandlerFunction func, void *client_data,
                                EControlContext ec);
void
ECsubmit_general_event (ECSourceHandle handle, void *event, int event_len,
                        EventFreeFunction free_func);
void
ECsubmit_general_typed_event (ECSourceHandle handle, void *event,
                        EventFreeFunction free_func);
void
ECsubmit_general_eventV (ECSourceHandle handle, IOEncodeVector eventV,
                         EventFreeFunction free_func);
```

Figure 6 shows a program which handles events in multiple threads. This example is written using the gen_threads package to access the thread library and calls `gen_pthread_init()` to bind those calls to the corresponding Pthread routines. The main program creates an event channel and forks two threads, each of which subscribe to the channel and events on private `EControlContext` values. `thread1_func` waits for events continuously using the `EControl_run()` function, while `thread2_func` polls (using `EControl_poll()` every 10 seconds. The main thread submits events to the channel at half second intervals. When this program is run it will print out "I'm submitting xxx" every half second. Immediately after event submission, the first thread is woken to handle the event and its handler is called. Every ten seconds, the second thread will poll for events and its handler will be called for all of its events (typically 20 events). Note that in this example, the call to `ECsink_subscribe_context()` is made by the main thread, but that is not strictly necessary. The subscribe call could just as easily have been done by the thread which would service the handler. Also note that the same event handler subroutine is used by each thread, but its output is customized using the `client_data` parameter in the subscribe call.

4.4 ECho Summary

ECho was initially developed as a data transport mechanism to support work in application-level program monitoring and steering of high-performance parallel and distributed applications[ES98]. In this environment, efficiency in transporting large amounts of data is of critical concern to avoid overly perturbing application execution. Because of this, ECho was designed to take careful advantage of CM and PBIO features so that data copying is minimized. Even typed event transmissions require the creation of only small amounts of header data and require no copying of application data. Also, because ECho transmits events directly to sinks, it naturally suppresses event traffic when there are no listeners.

However, as in many other situations using event-based communication, program monitoring can produce large numbers of events which may overwhelm both the listeners and the intervening networks. This can be particularly frustrating if the listener is not interested in every byte of data that it receives. Unwanted events waste network resources in carrying the data, cause unnecessary perturbation to the application sending the data, and waste compute time for the listener who has to be interrupted, read, unpack and discard events he would rather not be bothered with. Using many event channels to subdivide dataflows is an effective and low-overhead of reducing unwanted traffic because listeners can limit their sink subscriptions to event channels carrying data that they want to receive. However, effective use of this technique requires the event sender have *a priori* knowledge of the appropriate subdivisions. The technique is also much more difficult to apply when

```
static void handler(void *event, int length, void *client_data, attr_list junk)
{
    printf("In the %s handler, thread (%lx), event is %ld\n",
           (char*)client_data,
           (long)thr_thread_self(), *(long *) event);
}

static void thread1_func(EControlContext ec)
{
    EControl_run(ec);
}

static void thread2_func(EControlContext ec)
{
    while(1) {  /* handle events every 10 sec */
        sleep(10);
        printf("Handling events in thread2\n");
        EControl_poll(ec);
    }
}

int
main()
{
    CManager cm;
    EControlContext ec, ec1, ec2;
    ECSourceHandle handle;
    EChannel chan;
    long a = time(NULL) % 100;

    gen_pthread_init();
    cm = CManager_create();
    ec = ECho_CM_init(cm);
    chan = EChannel_create(ec);
    handle = ECsource_subscribe(chan);

    ec1 = EContext_create();

    ECsink_subscribe_context(chan, handler, "first", ec1);
    thr_fork((void_arg_func)thread1_func, (void*)ec1);

    ec2 = EContext_create();
    ECsink_subscribe_context(chan, handler, "second", ec2);
    thr_fork((void_arg_func)thread2_func, (void*)ec2);

    while (1) {
        long *event = malloc(sizeof(long));
        *event = a++;
        printf("I'm submitting %ld\n", *event);
        ECsubmit_general_event(handle, event, sizeof(*event), free);
        CMsleep(cm, 1);
    }
}
```

Figure 6: Event handling in specific threads.

a listener's definition of "unwanted event" depends upon the event content.

ECho's *Derived Event Channels* allow sink-specified event filtering, and even event data reduction, to be applied on the source end of event transmission. Performing these calculations at the source can be a win-win situation, reducing costs for both the sender and receiver and reducing bandwidth requirements on the intervening networks. The next section describes the Derived Event Channel abstraction and the critical role of dynamic code generation in performing these calculations in an efficient way in a heterogeneous environment.

5 Derived Event Channels

5.1 General Model

Consider the situation where an event channel sink 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. Considerable network traffic could be avoided if we could somehow manage to transmit only the events of interest. One way to approach the problem is to create a new event channel and interpose an event filter as shown in Figure 7.

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 is a nice solution in that it 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 there is more than one source subscribed to the original event channel and those sources are not co-located, as shown in Figure 8, then we still have raw events traveling over the network from Process A to Process B to be filtered.

The normal semantics of event delivery schemes do not offer an appropriate solution to the event filtering problem. Yet it is important problem to solve because of the great potential for reducing resource requirements if unwanted events can be suppressed. Our approach involves extending

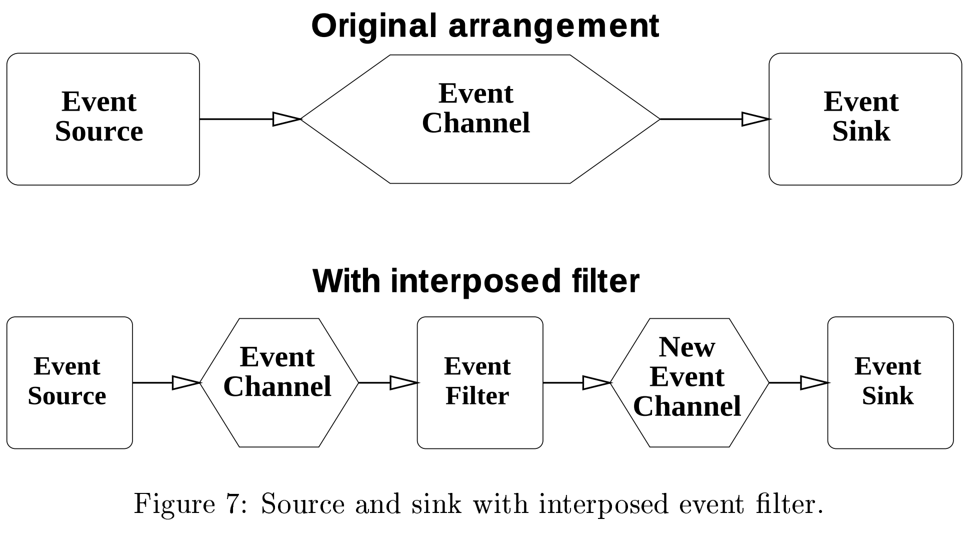

Figure 7: Source and sink with interposed event filter.

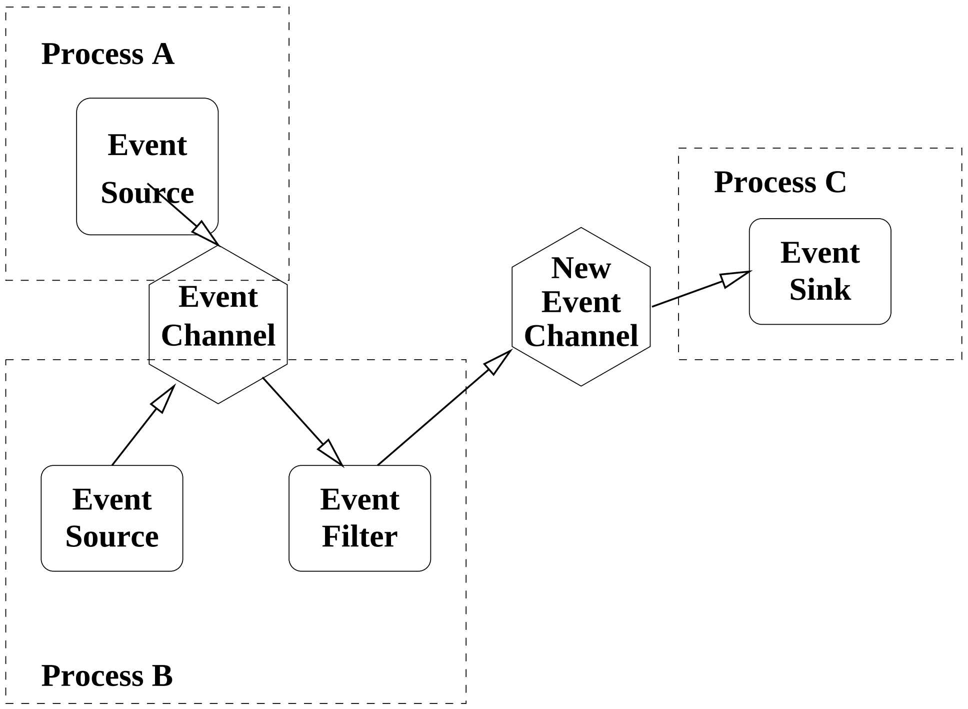

Figure 8: Filter with more than one source.

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 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 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. any of the sources in the original traffic, network traffic between those elements is avoided entirely. Figure 9 shows the logical arrangement of a derived event channel.

5.2 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 sink 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[COSNot] and in Siena[CRW98]. This approach facilitates efficient interpretation, but the restricted language may not be able to

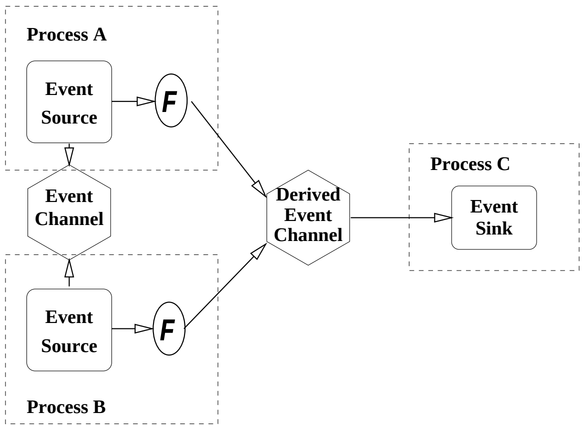

Figure 9: A derived event channel and function F moved to event sources.

express the full range of conditions that may be 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 sink 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 impacts efficiency and requires a potentially large interpreter environment everywhere event channels are used. Given that many useful filter functions are quite simple and given our intended application in the area of high performance computing we rejected these approaches as unsuitable. Instead, we consider these and other approaches as a complement to the methods described next.

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 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. Currently it supports the C operators, `for` loops, `if` statements and `return` statements. Extensions to other language features are straightforward and several are under consideration. Please contact the author for more information.

E-Code's dynamic code generation capabilities are based on Icode, an internal interface developed at MIT as part of the 'C project[PEK96]. Icode is itself based on Vcode[Engler] 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 and x86 processors[5]. 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.

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

```
EChannel EChannel_derive(EControlContext ec, char *chan_id, char *filter_function);
EChannel EChannel_typed_derive(EControlContext ec, char *chan_id, char *filter_function,
                               IOFieldList field_list, CMFormatList format_list);
```

Figure 10: Derived event channel API.

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, the input event. If the function returns non-zero it is submitted to the derived event channel, otherwise it is filtered out. Event filters may be quite simple, such as the example below:

```
{
    if (input.level > 0.5) {
        return 1; /* submit event into derived channel */
    }
}
```

When used to derive a channel, this code is transported in string form to the event sources associated with the parent channel, is parsed and native code is generated at those points. The implicit context in which this code evaluated is a function declaration of the form:

`int f(`⟨*input event type*⟩ `input)`

where ⟨*input event type*⟩ is the type associated with the parent channel.[6] The API for channel derivation is shown in Figure 10. 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 can also be associated with it.

While this basic support for event filtering is a very powerful mechanism for suppressing unnecessary events in a distributed environment, ECho also supports derived event channels where the event types associated with the derived channel is 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(`⟨*input event type*⟩ `input, `⟨*output event type*⟩ `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.

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 [KSSTA96]. Further, consider a sink 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 sink to receive the entire event and do its own data reduction we could save considerable network resources 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 following E-Code function:

```
{
    int i;
    int j;
    double sum = 0.0;
    for(i = 0; i<37; i= i+1) {
        for(j = 0; j<253; j=j+1) {
```

[6]Since event types are required, new channels can only be derived from typed channels.

```
typedef struct _ECdata_struct {
    IOFieldList data_field_list;
    CMFormatList data_subformat_list;
    void *initial_value;
}ECdata_struct, *ECdata_spec;

EChannel EChannel_derive_data(EControlContext ec, char *chan_id, char *filter_function,
                              ECdata_spec data_spec);
EChannel EChannel_typed_derive_data(EControlContext ec, char *chan_id,
                                    char *filter_function, IOFieldList field_list,
                                    CMFormatList format_list, ECdata_spec data_spec);

ECDataHandle EChannel_data_open(EChannel channel, IOFieldList data_field_list,
                                CMFormatList data_subformat_list);
void EChannel_data_update(ECDataHandle data_handle, void *data);
```

Figure 11: Channel Data in the derived event channel API.

```
            sum = sum + input.wind_velocity[j][i];
        }
    }
    output.average_velocity = sum / (37 * 253);
    return 1;
}
```

performs such an average over atmospheric data generated by the atmospheric simulation described in [KSSTA96], reducing the amount of data to be transmitted by nearly four orders of magnitude.

5.3 Parameterized Derivation

Currently derivation functions are simple functions of their input events. However, there are some obvious ways where more powerful functions could be valuable. Consider the situation where a sink wants a filter function based on values which change (hopefully more slowly than the event stream they are filtering). A simple example might occur in a distributed virtual reality application using event channels to share participant location information. Rather than sending location information as fast as the network allows, a more intelligent system might use derived event channels to turn down the update rate when the participants are not in sight of each other or merely distant. However, these conditions obviously change over time. One could periodically destroy and re-derive channels with updated filter functions, but a more straightforward approach would be to associate some state with a derivation function and allow it to be updated by the originator.

This facility is provided in ECho through several channel derivation functions that allow **channel data** to be associated with the derived channel. This data block is accessible to the derivation functions as a persistent structure named `channel_data`. The contents of the structure, like those of events, are defined through PBIO-style definitions at the time the channel is derived. Initial values for the structure must be provided when the channel is derived. Thereafter values for the structure can be updated through the `EChannel_data_update()` call. This call requires a `ECDataHandle` value, which can be obtained through `EChannel_data_open()`.

5.4 Proto-Channels and Derivation

Another innovative ECho extension is the Proto-Channel facility. Consider an application which has information that it wishes to make available on a more or less continuous basis, either in the

```
ECproto ECproto_create(EControlContext ec, ecl_parse_context context);
char *ECproto_id(ECproto proto_chan);

EChannel ECproto_derive_periodic(EControlContext ec, char *proto_id, char *filter_function,
                                 IOFieldList field_list, CMFormatList format_list,
                                 int microsec_period);

EChannel ECproto_derive_pull(EControlContext ec, char *proto_id, char *filter_function,
                             IOFieldList field_list, CMFormatList format_list);
```

Figure 12: Proto-channel API.

form of variables or function results. The application might make this information available via event channels, but to do that it must decide on a form that the events will take and a frequency with which it will be made available. This is less optimal than the ideal situation where the receivers of the data would specify what they wanted to receive and the timing at which it was made available.

Proto-channels address precisely this situation. A proto-channel is not an event channel in and of itself, but is an association of variables and functions from which channels can be derived. Like event channels, proto-channels have global IDs through which they can be accessed remotely. However, events are never submitted to them. Instead those who wish to access the data provide a derivation function which constructs an appropriate event from the available variables and functions. They also specify that the circumstances under which the function is to be run. The current API provides for the derivation functions to run either periodically or on demand via the event pull mechanism.

Proto-channels are created using the `ECproto_create()` call. In addition to a `CManager` parameter, this call takes an `ecl_parse_context` parameter which defines the variables and functions which will be available to derivation functions. The construction of `ecl_parse_context` values is described in [EisenDCG]. Channel derivation is accomplished through `ECproto_derive_periodic()` and `ECproto_derive_pull()`. The former call specifies a period at which the derivation function execute.[7] The latter call assumes that any use of the derived channel is by active sinks using pull-mode event requests. The proto-channel API is given in Figure 12.

6 Miscellaneous

There is a Java based version of ECho called JECho[JECho]. It was developed by Dong Zhou.

Additional information on ECho, including a porting guide for users of the DataExchange-based ECho version 1, can be found at http://www.cc.gatech.edu/systems/projects/ECho.

References

[CRW98] 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.

[7]Actual periodicity is not guaranteed and is subject to scheduling and CM service considerations at the proto-channel.

[EisenDCG] Greg Eisenhauer. Dynamic Code Generation with the E-Code Language. Unpublished. /users/e/eisen/ecl/doc/ecode.ps.

[Eisen94] 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).*

[ES98] 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.

[Eisen00CM] Greg Eisenhauer. The Connection Manager Library. http://www.cc.gatech.edu/systems/projects/CM

[Engler] 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.

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

[COSNot] Object Management Group. Notification service. http://www.omg.org, Document telecom/98-01-01, 1998.

[KSSTA96] 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.

[MS95] B. Mukherjee and K. Schwan. Implementation of scalable blocking locks using an adaptive threads scheduler. In *International Parallel Processing Symposium (IPPS)*. IEEE, April 1996.

[PEK96] 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.

[RSYJ97] 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.

[JECho] Dong Zhou. JECho Project Homepage. http://www.cc.gatech.edu/~zhou/jecho