# BEE: A Basis for Distributed Event Environments (Reference Manual)

Bernd Bruegge

3 November 1990
CMU-CS-90-180 3

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

**Abstract**

BEE is a portable platform for building heterogenous distributed event environments. An important feature is the dynamic connection of client programs to monitoring tools which facilitates flexible monitoring of network programs at runtime. It also supports user defined event classes which can be used by implementors to build complex event systems as well as by application programmers who need to write customized monitors.

We first introduce the event processing model used by BEE and present the user's view, describing the instrumentation of network programs and a set of standard event interpreters providing graphical views based on X11. Some performance measurement results are given to demonstrate the cost associated with BEE. The rest of the document describes the functional specification of BEE.

BEE has been implemented for a variety of platforms, communication systems and languages. It is currently available on NECTAR, a network of workstations connected by optical fibers with a maximal throughput of 100 Mbit/sec and on UNIX. The UNIX implementation has been ported to several machine architectures (Sun, Vax and Cray-YMP), supporting the instrumentation of C and Ada programs.

This research was supported in part by the Defense Advanced Research Projects Agency, DARPA/ISTO, under the title "Research on Parallel Computing," ARPA Order No. 7330, issued by DARPA/CMO under Contract MDA972-90-C-0035; and in part by the National Science Foundation and the Defense Advanced Research Projects Agency under Cooperative Agreement NCR-8919038 with the Corporation for National Research Initiatives. The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of DARPA, NSF, CNRI or the U.S. Government.

**Keywords:** Network monitoring, Real-time systems, Distributed monitoring, Performance monitoring, Distributed systems.

# Table of Contents

# List of Figures

# 1 Introduction

With the advance of parallel machines application programmers take a new look at their programs to achieve performance improvements by parallelization. However, with the parallelization comes a higher level of complexity because the program is now distributed over many processors and the computation is no longer as visible as on a single processor. Many new questions arise: Are some of the processors idle, should the load on the processors be balanced better? What are the bandwidth requirements on the communication subsystem, what is the cost of communication, how are the processes communicating with each other? Can we use the monitoring information dynamically to improve the performance of the program? With these questions comes the need for new monitoring, debugging and performance evaluation tools.

Instead of providing a multiplicity of different monitoring tools, we believe it is more desirable to provide a common basis on which these tools can be built. BEE is a platform for event processing on top of which users can build monitors, debuggers and performance evaluation tools. We identify a small set of functions common to event processing and call this set the *event kernel*. An important part of the event kernel is the definition of an event protocol, which specifies the format of events exchanged between client program and event processing system. The event protocol makes the event kernel highly portable for different communication protocols.

BEE views the execution of a distributed system as the generator of streams of events [3]. In general, event based systems increase the execution time of the client program because of the overhead spent in event generation. They are also space inefficient because of the number of events to be stored and runtime inefficient because of the slowness of many existing I/O devices. For example, networks such as Ethernet are too slow to support remote event processing with reasonable event latency (the time between generation and interpretation of the event).

To reduce the execution overhead, event based systems usually provide *filters* to prevent the creation and/or interpretation of unnecessary events [18]. Another way to reduce the overhead is to delay the event interpretation to a postprocessing phase. This is used in many monitoring systems, for example [12, 21], but it is unacceptable for monitoring problems where the information is needed at runtime [4]. Event processing systems providing *runtime monitoring* often separate event generation and analsis by sending events to remote monitors, which may also combine event streams from different nodes [7, 10, 15, 16].

Most of the existing event processing systems provide a rather static connection of the client to the event processing system. However, when monitoring distributed programs, desired event interpretations can often only be formulated at execution time. BEE tries to provide a higher degree of flexibility: Users are able to customize the monitoring process by connecting client programs and monitoring tools at runtime. This flexibility allows the definition of user defined views that can be dynamically modified during the execution.

The document is organized as follows. In Section 2 we present the notational framework of BEE followed by a description of the event processing stages and a discussion of how they can be parallelized in Section 3. Section 4 demonstrates how BEE can be used to build a wide spectrum of event processing systems, ranging from simple single process systems to distributed event systems providing multiple views of the execution. BEE was designed with several portability goals in mind and Section 5 portrays some of the portability problems encountered during the implementation. Section 6 discusses the cost introduced by event sensors and evaluates the runtime performance of BEE.

BEE is currently available on NECTAR [2] and on UNIX. Section 7 describes how to use BEE under NECTAR, in particular how to instrument a client program, how to start event interpreters and how to connect clients with event interpreters and to build customized event interpreters. Section 8 contains the functional specification of the event kernel and Section 9 is a description of the underlying event protocol. Appendix I contains a summary of the

functional specification.

I would like to acknowledge the contribution of several people who participated in the design and implementation of BEE at various stages. Peter Steenkiste provided many comments during the initial design. Johannes Mann from Siemens Corporation wrote BEE's X library interface and provided the first generation of X based event interpreters in C++. Frank Walzer from Siemens Corporation improved the performance of the event interpreters and performed the measurements under NECTAR. Marco Gubitoso and Hiroshi Nishikawa were the first application users and provided many valuable comments. Mike Browne ported BEE to TCP/IP. Mario Barbacci from the Software Engineering Institute encouraged BEE's port to the Vax and contributed the language interface for Ada. Jim Lewis from the Pittsburgh Supercomputing Center helped to measure BEE's performance on the Cray-YMP.

## 2 BEE's Event Processing Model

A *(atomic) event* E is a 7-tuple (**Class, Attribute, Node, Process, Thread, Timestamp, E_ID, Variant**). Class partitions the event space into equivalence classes. The **Attribute** indicates whether the event is an activation (E_ACT), a termination (E_TERM), a point (E_POINT), an aggregate event (E_AGGREGATE) or a probe (E_PROBE). Events attributes denote the activation and termination of an *event range*, respectively, events of type E_POINT denote a interesting point in the execution of the client. Event aggregates denote the accumulation of several atomic events. **Node, Process** and **Thread** identify the location where the event occurred. **Node** identifies the workstation, **Process** is the client name assigned by the operating system and **Thread** provides a unique identification for threads in multithreaded applications.

**Timestamp** is the value of a clock when the event occurred. In general, events from different nodes do not have access to the same clock and sorting event streams generated by clients on different nodes is a significant problem. BEE provides a function that allows to sort time stamps but it assumes constant communication delay and the absence of clock drift. Practical schemes for synchronizing clocks at different locations are inherently approximate due to the unpredictability of communication delays [8]. The resolution of timestamps is another problem [15]. The Cray clock provides nanosecond resolution, but for portability reasons, time stamps in BEE have only microsecond resolution.

The event descriptor E_ID identifies the event sensor that caused the occurrence of the event. BEE supports predefined event classes such as the execution of a procedure, as well as user-defined events to mark important milestones in the execution of the application. **Variant** contains data that depend on the specific event class. For example, the event class E_PROCEDURE has an additional field denoting the name of the procedure that triggered the event[1].

Event processing in BEE is based on four activities operating on events: event sensing, event generation, event handling and event interpretation (figure 1).

Events are detected by *event sensors* [18] which are inserted into the client program. The implementation of event sensors depends on the event class. If the instrumentation of a client with event sensors requires more space than the uninstrumented client, the sensor is invasive, otherwise noninvasive. For example, a break instruction is noninvasive if it is done by code replacement. BEE is an invasive environment, because event sensors are macros inserted by the programmer and expanded by a preprocessor (See Section 8.2.).

The *event generator* collects the components of the event, that is the class, attribute, timestamp and class

[1] See Section 9.3, page 52 for a detailed description of the event components.

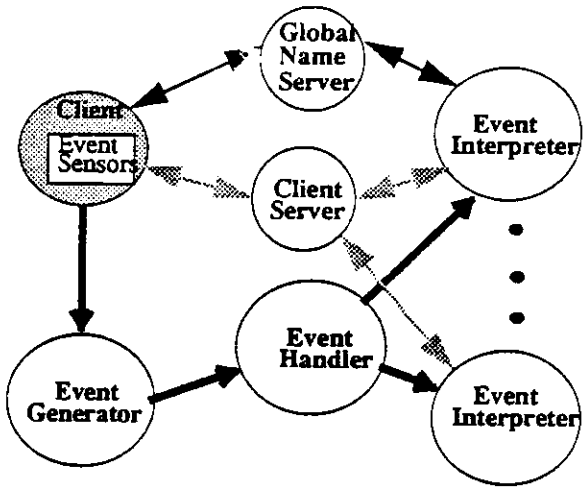


**Figure 1:** BEE's Event Processing Model

dependent variants and sends it to the event handler.

The *event handler* manages the event table, which is an array of the defined event classes. Each event class can be associated with an ordered list of event interpreters. For an event of a given class, the event handler scans the list and sends the event to each attached event interpreter.

An *event interpreter* consists of five components EI = (N, I, E, F, P). N is the name of the service under which the event interpreter is known in the network. The `Init` function I is called immediately after the event interpreter is attached to the client program. The `Every` function E is called whenever the client generates an event and the *Predicate* function P evaluates to TRUE. The `Final` function F is called when the client terminates or when the event interpreter is detached from the client.

The *client server* allows event interpreters and other programs to request information about the client program, as well as to alter the execution and the state of the client. At startup, each client program creates a unique port called the *client server port*. Requests that are sent to this port are handled by the client server, which executes in the same address space as the client program. BEE supports different types of requests. The client server can be used to retrieve the names of variables defined in the client program; this allows events to use a concise representation for names, and to have the event interpreter retrieve naming information from the client, when needed. The functionality provided by the client server is similar to that provided by `ptrace` in UNIX.

Monitoring plays an important role when developing network programs, but it is not always possible to know in advance what aspect of the execution should be monitored. The network monitoring environment must therefore allow the user to change the monitoring setup during execution, in fact, it should be able to provide feedback information back to the client program [4]. The client server allows users to change the flow and use of events during execution: the user can send requests to the client server port to activate or deactivate event classes, change the frequency of event generation, or to add event interpreters that give the user a different view of the application.

When the client server is started, it enters its client server port in a global name server (see figure 1). Event interpreters and other programs can retrieve the port identifier based on the name of the client at any time during program execution. If the runtime system supports multiple threads, the client server is a separate thread in the client program. If threads are not available, it is implemented as an interrupt handler where each event interpreter causes an interrupt of the client execution.

The execution of a client with one or more attached event interpreters is called an *event configuration*. Because event interpreters can be attached dynamically to a client, the character of an event configuration depends on the time when the event interpreters are attached to the client. We distinguish two event configurations: If the client starts up first and the event interpreter is attached later, we call this an *unplanned event configuration* otherwise the event configuration is *planned*. BEE supports both types of event configurations.

## 3 Efficiency in Event Processing

The performance of an event processing system can be characterized by *client overhead* and *event latency*. Client overhead, also called monitoring perturbance, is the additional cost experienced by the client in terms of space and execution time as a result of the instrumentation. Client overhead is mainly a function of the event rate and the cost to assemble and pass events to the event processing system before the client can proceed its execution. The assembly can be more or less expensive depending on the underlying operating system. For example, in UNIX it takes a system call to get the current time for the timestamp, whereas on NECTAR the time is available in a register. An event processing system with low impact on the execution time of the client and few demands on memory resources has a high client efficiency.

Event latency is the average response time of the event interpreter once an event sensor is encountered. The event latency is a function of three parameters: the cost per event, the network latency of the underlying communication system and the event service time, which is the time used by the event interpreter to process the event.

Given a fixed event rate and event service time, client overhead and event latency are inverse to each other: the smaller the client overhead, the larger the event latency. To select the best combination of client overhead and event latency, BEE event interpreters can be attached in various ways to client programs. In a *local* event interpreter, `Init`, `Every` and `Final` are functions located in the address space of the client program. A local event interpreter provides the lowest possible event latency at the cost of high client overhead. At the other end of the spectrum is the remote event interpreter, which executes on a separate computation node. If a client is connected to a remote event interpreter, `Init`, `Every` and `Final` are remote procedure calls. A remote event interpreter provides lower client overhead at the expense of increased event latency.

To choose an overhead/latency combination between the ends of this spectrum, a local and remote event interpreter can work together as follows: The `Every` function of the local event interpreter accumulates individual client events. When a specified threshold is reached, the local event interpreter calls its `Final` function, which sends the events as an aggregate event to the `Every` function of the remote event interpreter.

BEE achieves event efficiency by parallelism in the event processing model itself and by filtering. In section 3.1 we describe how the event processing stages in the model could be parallelized and how it is actually done in BEE. In section 3.2 we describe how filters minimize the number of events that actually have to be generated or sent to an event interpreter.

### 3.1 Efficiency by Parallelism: Design Choices

Distributed systems provide the programmer with the challenge to detect parallelism in applications and achieve faster run times by mapping the application code onto many processors. In the following we discuss possible ways to achieve parallelism in an event processing system itself and present the rationale for the parallelization used in BEE. Candidates for parallelism are the four activities event sensing, event generation, event handling and event interpretation.

To collect detailed sensor information, with minimal impact on the application, other researchers have

implemented hardware or hybrid monitors [9, 13, 14, 22]. This approach is not taken in BEE for portability reasons. Without adequate hardware support for event sensing it is too expensive to separate this activity from event generation. BEE therefore places event sensing and event generation in the same process.

Event generation and event handling can be separate processes communicating via an *event handler buffer*. The generator drops the event into the event buffer from where it is received by the event handler. Again, without hardware support this decomposition is not advisable on a single processor, because it introduces only scheduling overhead. If the target machine is a multiprocessor with shared memory, event generator and event handler can be tasks executing on different processors accessing the event buffer in shared memory. A distributed implementation with two different nodes should not be attempted if the communication subsystem cannot keep up with the event generation rate. In the current implementation of BEE, event generation and event handling are handled by the same process, but in the NECTAR implementation we are looking into a separation where the event handler is placed on the CAB coprocessor.

Event interpreters are ideal candidates for separate processes. Often they provide a graphics user interface for visualization which requires a high computation rate impacting the client efficiency if done on the client processor. The separation has another advantage: The event interpreter can be written in a language different from the client language. Usually the client program has to be as efficient as possible, whereas event interpreters need to be interactive, flexible and allow for rapid prototyping. Whenever an event occurs, the event handler sends the event as a message to the event interpreter via the *EI buffer*. If a computation requires multiple views, the event handler sends the message containing the event to all attached event interpreters. Figure 2 shows a decomposition for a single client with 3 attached event interpreters using separate processes for handling and interpretation of events.

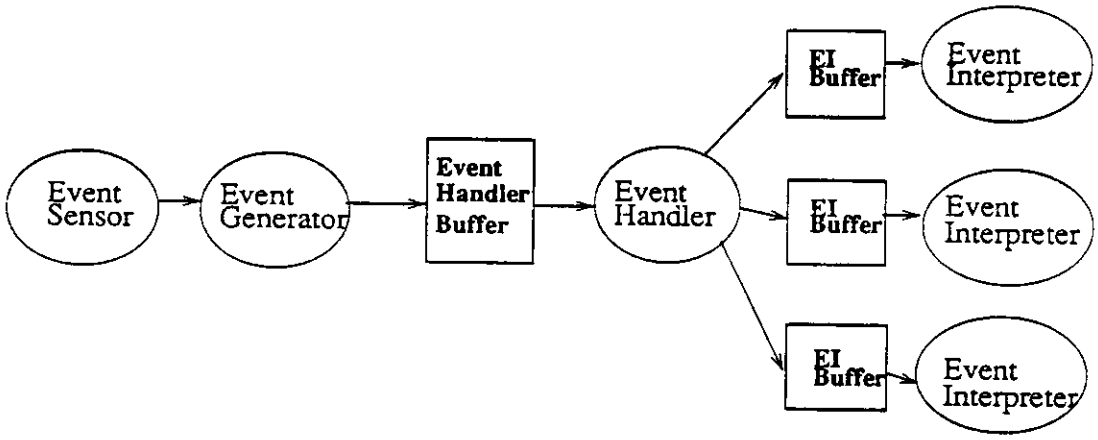


**Figure 2:** Parallelization of BEE with 3 Event Interpreters

### 3.2 Efficiency by Filtering

Event filters reduce the communication bandwidth imposed by the event processing facility. The challenge is how to avoid too much computation on the client side to determine whether an event should be filtered: Filters increase event efficiency, but filter computations decrease it. The right balance of filters and filter computation is a matter of experimentation and can often only be decided by the user. BEE therefore provides filters for each event activity whose use can be controlled by the user.

At the event sensing stage, event sensors can be passive, dormant or active. Ideally, passive sensors should no influence the behavior of the client program. A dormant event sensor is similar to an event probe [18]. The event

generator is called only when the sensor is encountered the first time. From then on its value must be requested by the event interpreter. When an active sensor is encountered, the event generator is always called. Thus BEE's event sensors support sampling as well as tracing of client programs [11]. The state of event sensors can be changed with the two functions `Event_Sensor_Control()` and `Event_Sensor_Filter()`. The first one enables or disables event sensors, the latter can change active sensors into passive filters whose name is not a substring of a specified string called the *global sensor filter*.

To filter events at the generation stage, BEE provides two kinds of event tables: a system wide event table and task specific event tables. The system event table is created at client startup time and initialized with the predefined event classes. Whenever a new client task is created, a task event table can be instantiated which inherits the currently defined event classes from the system event table. Filtering by task specific event tables is controlled with the function `Event_Table_Inheritance()`. If event table inheritance is disabled, all events are processed via the system event table. If it is enabled, events are processed on a task basis: If the event is generated inside a task, it is generated in the context of the associated task event table, otherwise the system event table is used. It is also possible to turn off event handling for individual tasks. If no event interpreter is associated with the current event table, no event is generated. This permits selective filtering of events on a task specific basis.

The event handler filter can selectively drop events or accumulate a number of events before sending them to the event interpreter. For example, an event handler filter can accumulate the events generated in a certain time, or send only the latest event generated in the last n seconds. The predicate function of an event interpreter also provides filtering at the event handler level. If the function evaluates to FALSE, the event will not be sent to the event interpreter.

Very expressive event detection languages with powerful filters such as [5] can be used to provide filtering at the event interpreter level. An example is a distributed composite event, where the event interpreter waits for the encounter of a certain event before disabling another one. Another possibility to filter events is abstraction, for example by combining lower level events to higher level events before presenting them to the user [3]. Facilities like these should be placed in remote event interpreters, because they require resources that significantly decrease the client efficiency. In fact, they often require knowledge from several clients and cannot be placed into the address space of a single client.

## 4 Event Configurations

BEE is a platform for building distributed environments. In the following we show how BEE can be used to build a wide spectrum of event processing systems, ranging from simple single process debuggers to distributed event systems providing multiple views of the execution.

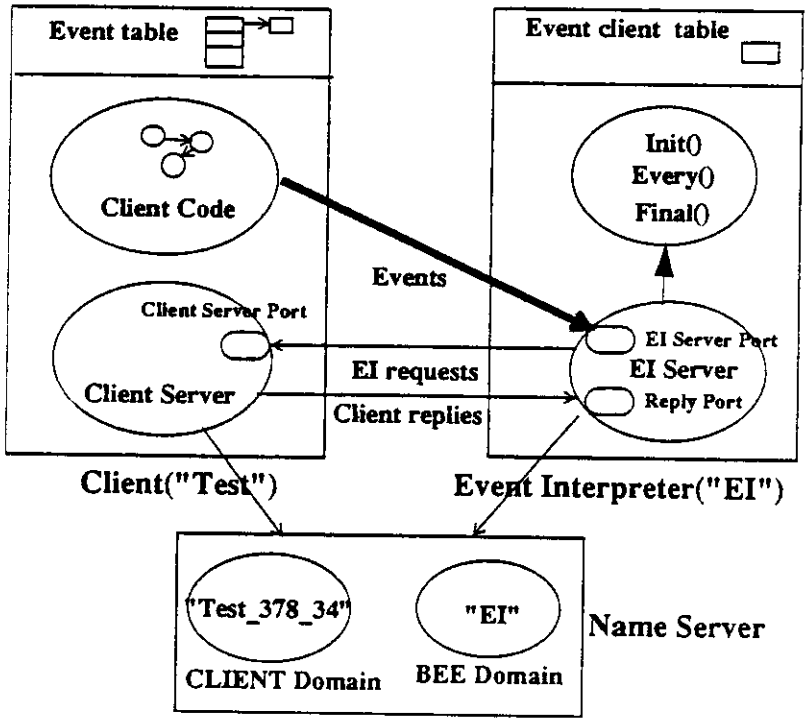


**Figure 3:** Basic BEE Architecture

Figure 3 shows the basic event configuration of a client `"Test"` connected to an event interpreter `"EI"`, each of them executing in their own address space. At startup of the client the *Client Server* is created. The *EI Server* is created when the event interpreter starts up. The client server as well as the EI server register themselves with the (global) name server using the `Event_Enter_Service()` function. The name entered by the client server uniquely identifies the client by its name and process id and the node id on which it is executing.

If the event configuration is planned, the client calls `Event_Lookup_Service("EI", "BEE")` which looks up the *EI server port* of `"EI"` in the name server domain `"BEE"`. The first event the client sends is a message of type E_REGISTRATION to the event interpreter, which enters the client into its *Event client table*. This is followed an event message of type E_INIT, which causes a call to the `Init()` function of the event interpreter. From then on, any event generated in the client is sent to the `Every()` function of the event interpreter, provided `"EI"` is enabled, attached to the event class and its predicate function evaluates to TRUE. When the client detaches from the event interpreter, it sends an event message of type E_FINAL() to the EI Server which calls the `Final()` function of the event interpreter. All this communication takes place on the connection labeled **Events** in figure 3 using a reliable message protocol.

Requests issued by the event interpreter are sent on a different connection and use a request/response protocol. These **EI requests** are sent to the client server port and are served by the associated client server. Provided with the EI request is a reply port to which the client server sends its **Client replies**. For example, if the event configuration is unplanned, the EI server gets in touch with the name server to locate the client server with `Event_Lookup_Service("Test_378_34", "CLIENT")` which returns the *client server port*. It then

attaches to the client with `Event_Set_Client_Port()` and sends the request `Event_Lookup_Interpreter("EI", "BEE")`.

The functionality described above supports the construction of arbitrary complex event configurations. In the following we show the most important configurations that can be built with BEE. In the figures a rectangle indicates a process (an address space), circles indicate modules or threads, in particular a circle marked C is an instrumented client and a circle marked EI is an event interpreter. Arrows between the circles indicate communication paths used by the event kernel.

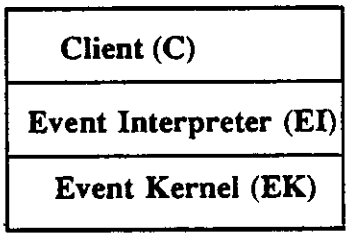


**Figure 4:** Local Event Interpreter Configuration

In the simplest BEE configuration, the event interpreter and the client share the same address space (see figure 4) and the event interpreter functions are called directly by the client. This architecture is used in many early debuggers. If the local event interpreter is a separate thread such as in figure 5, it models the architecture used by the PARASIGHT debugger [1].

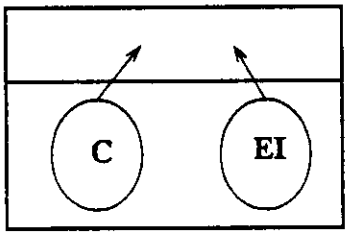


**Figure 5:** Shared Memory Configuration

Figure 6 shows a client connected to several event interpreters, each of them tapping on the same event stream, but providing different views of the behavior [17, 19]. The different event interpreters can either be on the same node or on different nodes.

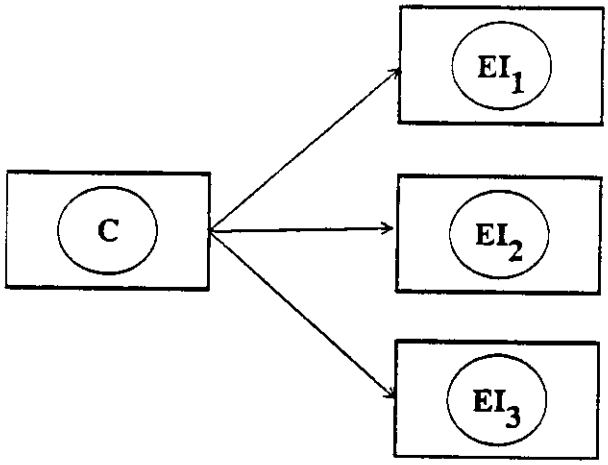


**Figure 6:** Multiple views of a client

A BEE configuration that could be used to implement a distributed debugger for shared memory applications is shown in figure 7. An event interpreter is connected to several clients each of them creating event streams. Because all clients have access to the same clock, the event streams can easily be totally ordered.

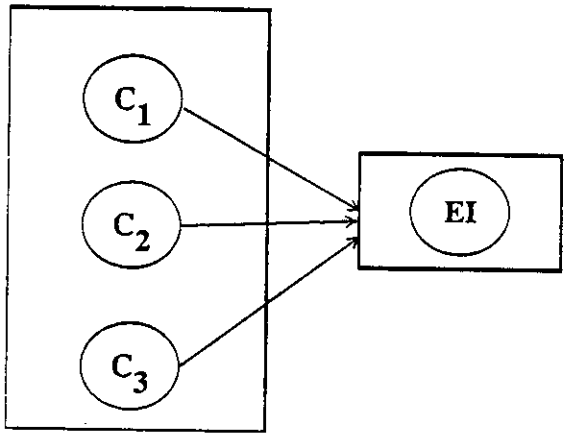


**Figure 7:** Monitoring a distributed client (shared memory)

Several clients executing in separate address spaces can also be attached to an event interpreter as shown in figure 8. This configuration is very important in a network environment since users want to get an overview of the activities in the entire system. An example is measuring the load on the network nodes for dynamic load balancing purposes. This configuration is frequently used by NECTAR applications.

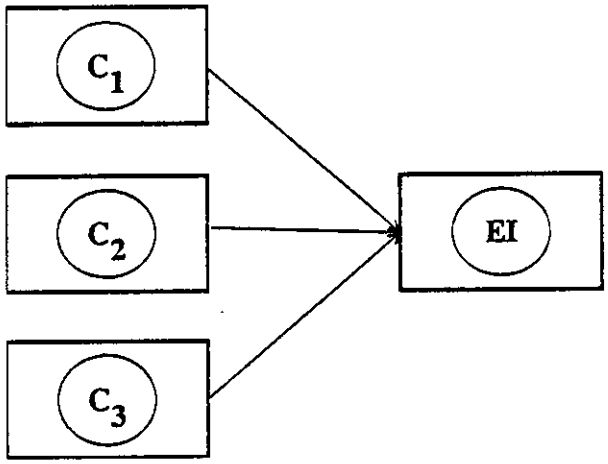


**Figure 8:** Network monitoring of multiple clients

For efficiency reasons, it is sometimes advisable to reserve a separate node in the system just for event interpretation. In this case it is possible to reduce the client overhead by having the client send the events to a central event interpreter which then *regenerates* the (filtered) events, causing them to be sent to other attached event interpreters (see figure 9). This 2-stage event handling reduces the communication bandwidth requirements but it increases the event latency.

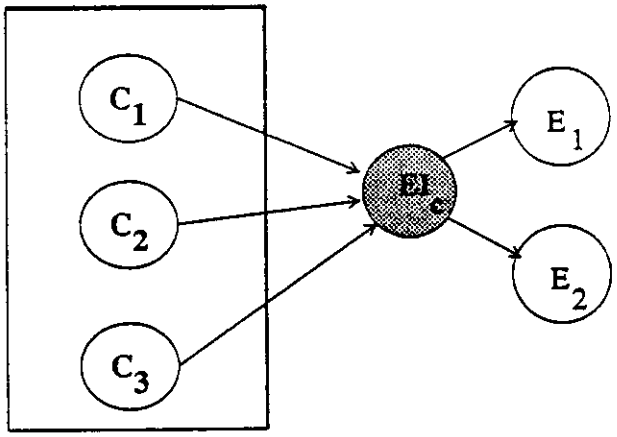


**Figure 9:** Multiple view configuration with central event interpreter

Figure 10 shows an event configuration with a central event interpreter and local event processing activities at each of the nodes. This configuration is used in Ogle, Schwan and Snodgrass's system [16]: Each node uses a "resident monitor" (local event interpreter) which collects and analyzes the monitoring information about processes executing on that node. The resident monitors report to a central monitor (remote event interpreter) executing on a network node with access to a monitoring data base.

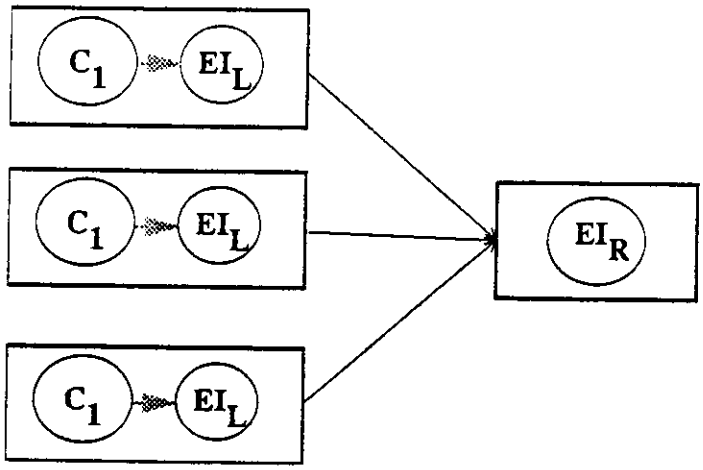


**Figure 10:** Distributed monitoring system with local event interpreters

### 4.1 The Dynamics of Event Reconfigurations

Because BEE permits the connection between a client and an event interpreter at runtime, it is possible that a service under a certain name has already been entered with the name server. The `lookup_mode` parameter in the event kernel function `Event_Enter_Interpreter()` specifies what to do in this case. The 'replace' mode states that the new service replaces the existing one. This mode is useful for debugging of event interpreters. Once an event interpreter is debugged, it should be entered into the name server with the 'error-if-exists' mode. In this mode, the event kernel issues an error message if an event interpreter is trying to enter an already existing service.

BEE also deals with the case where the client or the event interpreter finish the execution in an abnormal way. In the following we discuss both of these cases and their implication on the name server consistency.

**Client Death**

When the client finishes execution in an orderly way, it sends a E_CLIENT_DEATH message to the event interpreter and the event interpreter deletes it from the event client table. However, if the client terminates abnormally and the message is not sent, the event interpreter must be notified in a different way. In the socket implementation, the event interpreter is waiting on a `read()`. If the `read()` returns with an abnormal result, the event kernel on the event interpreter side assumes that the connection is no longer open, assembles a fake death message and passes it up to the event interpreter. In the Nectar implementation, the event interpreter issues a status call on the client server port at regular times. If the port no longer exists, it assumes that the client has been terminated abnormally.

In the case of an abnormal client death, the name of the client event server must also be deleted from the name server domain. It cannot be assumed that the client can do this. The event interpreter therefore asks the name server to delete the name of the client server (which was passed to it in the client registration message).

As far as the event sensors registered on the behalf of the client are concerned, the event interpreter has two choices. In the first choice, event sensors are kept around. This is useful for multiple client runs, where the user wants to compare the results of several runs. The other choice is to delete the sensors from the event interpreter tables and - if a grapical display is used - from the window the next time the display is updated.

**Event Interpreter Death**

An abnormal event interpreter death is more problematic. Again, the name of its service must be deleted from the name server domain, but no client should have the right to do this. Another issue is how the client recognizes that a connection is dead. The NECTAR implementation currently does not provide a solution to this problem. In the socket implementation the `write` operation returning abnormally indicates a dead connection.

There are two choices what to do when the client discovers a dead connection. One possibility is to no longer send events to the service. Another one is to reestablish the connection with the event interpreter. This involves two problems: First, the client must lookup the service at the name server again. Once the new connection is established, the second problem arises: Events must from now on be sent to the new service, but in general a lot of monitoring information has already been sent to the old one which is no longer available.

It is not clear for what kind of event interpreters a client should try to set up a new connection. In fact, it is not clear whether this is useful at all. If at all, a client should try to reestablish the connection to event interpreters that keep only minimal state. Event interpreters with minimal state are load meters, load balancers and debuggers.

If client reconnectivity is allowed the event interpreter must be prepared to get events in an unexpected way and send requests to the client server to get the missing information. There are some useful cases, where reconnectivity is desirable. For example, when replacing a buggy debugger by a better version, the client might have run for a long time and the lost state might consist of a few breakpoints which can easily be redefined. Event interpreters measuring network loads are also candidates for reconnectivity. Event interpreters with extensive state such frequency counters or time profilers should not be considered. When a time profiler dies, it makes no sense to reestablish the connection, because the times collected up to the point of failure are lost. Reconnection would lead to inconsistency if the client has already sent the activation of an event range before the failure. In this case the client should be restarted. In the current version of BEE, the death of an event interpreter is recognized, all connections from the client to that event interpreter are deleted, a warning message is printed and the client continues its execution.

**Name Server Consistency**

Because of the possibility of crashing clients and event interpreters, the name server might offer services which no longer not exist. Before actually giving out a service as the result of a request, the name server therefore sends a message to the client server port (which is passed as a parameter of the name server enter request). Only if the event client server is answering, does the name server assume that the program can be made part of an event configuration. Otherwise the service is deleted from the name server domain.

## 5 Portability

With the easy scalability of networks, it is almost inevitable that nodes become heterogenous even if the network is initially configured as a set of homogenous nodes. Furthermore, with the advance of high-speed networks, new protocols are constantly being developed that utilize the available bandwidth better than existing protocols. Finally, the separation of event generation and interpretation encourages the programmer to write client and event interpreters in different languages employing different compilers, runtime systems and operating systems. In fact, the client program itself can be a network program written in several languages. Several portability/efficiency tradeoffs were made in BEE which are discussed in the following.

Exchanging data between heterogenous nodes is a well known problem because architectures differ in their storage byte order of data as well as in their word size. In "little endian" architectures such as Vax computers, numbers are stored in byte swapped order, in "big endian" architures such as Cray machines and Sun workstations, the address of an integer is the address of the high-order byte of the integer. Even though TCP/IP defines a network order for 32 bit integers, network programs assuming a particular byte size for data are not portable, because on a Cray, for example, integers can be 24, 48 and 64 byte quantities.

To allow the exchange of event messages between client and event interpreters understood by all host machines, BEE's event protocol contains an exchange standard consisting of several message formats: an Ascii format and a set of host formats. In the Ascii Format all the components of an event message are represented as text. This format is

highly portable and used when exchanging messages between heterogenous nodes, for archiving events and for BEE's event replay facility. The host formats are an attempt to strike a balance between portability and efficiency. If a client and an event interpreter are compiled by the same compiler and execute on the same machine they can exchange messages in the same host format. To determine whether both parties can indeed share a format, the event protocol contains a negotiation part when the client attempts to connect to the event interpreter. As a result of the negotiation the client knows which format it can use. This information is stored in the event table maintained by the event handler, because on the client side it is the only event processing stage that needs to have access to this knowledge once the connection is established.

For portability reasons, the event protocol specifies the exact layout of the Ascii format, but none of the host formats. Host formats are determined de facto by the compiler's allocation strategy for structures. Being in a host format means only that the event message is represented as a binary structure. Currently BEE supports 3 host formats for Sun, Vax and Cray, but new host formats can be added easily.

The event protocol is connection oriented, because we can safely assume that a client wants to send more than one message to an event interpreter. However, with the advance of high-speed communication systems communication protocols might quickly become obsolete and need to be replaced by versions that are more efficient for the new generation of networks. BEE's use of communication system primitives is completely encapsulated in a small set of functions defined in the event protocol (See Section I.11). Only these functions need to be reimplemented when Bee is ported to another communication subsystem and this can be done very quickly. Originally Bee was implemented on Nectarine [20], the communication interface to NECTAR. The port to TCP/IP using Berkeley sockets took less than a week, clearly demonstrating Bee's high communication system portability.

Another problem is the ability of the event interpreter to send requests to the client server while the client application is running. This feature is important for getting needed information from the client as well as for efficiency reasons, because a dormant event sensor causes much less overhead on the client program than an active one. Operating systems provide different primitives for a process trying to asynchronously connect to another program. Several efficiency/portability tradeoffs are possible and two schemes have been implemented in BEE. In the first scheme the client server is implemented as a thread waiting on a `receive()` and event interpreters send client requests by sending a message to a port owned by the thread. The client server thread serves the request, sends the reply back to the event interpreter via a reply port indicated in the request and waits on `receive()` again. This scheme is very well suited for multiprocessor architectures, because event interpreter requests can be handled by the client server simultaneously while the client application thread executes undisturbed on another processor.

In the second scheme, the client server is implemented as an interrupt handler. In this implementation the event interpreter request causes a signal which interrupts the client program and invokes an event client signal handler. The event client signal handler then figures out from which port the signal came from and calls an event kernel function that processes the request. On a multiprocessor, the interrupt handler is clearly less efficient than the thread implementation, but many runtime systems do not support multiple threads and on a single processor architecture an interrupt might cause less overhead than a thread context switch. The current version of BEE uses SIGIO as signal to the client server, which can cause portability problems if this signal is intercepted by the client program.

# 6 Bee's Cost

BEE is an invasive event environment: it introduces overhead in terms of space and runtime. A major design goal of BEE was to keep the overhead as minimal as possible so that client programs can always be executed with instrumentation. Remote event interpreters provide one way to keep the overhead small, because the event interpretation is taking place on a different host. The other key to client efficiency is an efficient implementation of event sensors. Assuming no hardware support, we looked at several event sensor implementations discussed in Section 6.1[2]. Section 6.2 is an experimental evaluation of BEE's performance.

## 6.1 Sensor Implementation

A very simple implementation of an event sensor name checks only if event processing is enabled and if yes, generates the event:

```
#define BEGIN(name)
  if (Event_on) {  \
     Event_Generate(E_PROCEDURE, E_ACT, "name");  \
     } \
 {
```

This is not a good sensor implementation: Events are generated for all event sensors whenever `Event_on` is TRUE which provides only very coarse filtering. Another disadvantage is that `Event_Generate` is called with three parameters.

We can decrease the calling overhead by associating a unique event sensor ID **_eid_** with each event sensor and pass the ID instead of the name. To speed up event sensors even more, we define event generators for each predefined event class. Thus the event class and event qualifier do not have to be passed for predefined events. This results in the function `Event_Activate_Procedure` which takes only the **_eid_** as parameter. Each event sensor ID is a static variable initialized to 0[3]. The first time the sensor is encountered, the name generator `Event_Register_Name` is called which generates the unique ID and enters the name into a sensor name table. Once the sensor is registered, only the function `Event_Activate_Procedure` has to be called.

```
#define BEGIN(name) \
  { \
   static Event_id_t _eid_ = 0; \
   if (Event_on) { \
    if (_eid_ == 0) Event_Register_Name("name",E_PROCEDURE,&_eid_); \
    Event_Activate_Procedure(_eid_); \
     } \
   {
```

This sensor implementation calls the event generator for each event sensor if event processing is enabled, so the runtime overhead is still considerable.

If we assume that **_eid_ < 0** indicates a passive event sensor - and by changing the semantics of `Event_Register_Name` to call `Event_Generate` itself when an event is registered - we can use the following definition:

---

[2]The discussion is in terms of the activation and termination of the predefined event class E_PROCEDURE (See Section 8.1.2) and uses macro definitions in C.

[3]This is a special feature of C (comparable to Algol's own with initialization) and is not available in other languages.

```
#define BEGIN(name) \
  { \
   static Event_id_t _eid_ = 0; \
   if ((_eid_ >= 0) && Event_on) { \
    if (_eid_ == 0)  Event_Register_Name("name",E_PROCEDURE,&_eid_); \
    if (_eid_ > 0) Event_Activate_Procedure(_eid_); \
    } \
   {
```

Note that in C, the conditional expression **((_eid_ >= 0 ) && Event_on)** evaluates already to FALSE if the sensor is passive. If the sensor is active, the event is generated. Otherwise the sensor is assumed to be uninitialized: the name of the sensor is registered and the event is generated. Thus `Event_Generate` is called only for active event sensors.

Another runtime optimization can be done by introducing a registration function `Event_Register_Procedure` for the predefined event class E_PROCEDURE:

```
#define BEGIN(name) \
   { \
     static Event_id_t _eid_ = 0; \
     if ((_eid_ >= 0) && Event_on) { \
        if (_eid_ > 0) Event_Activate_Procedure(_eid_); \
        else \
          Event_Register_Procedure("name", &_eid_); \
        } \
      {
```

This macro is used for procedure sensors in the implementation of BEE. It requires the evaluation of one boolean variable for passive event sensors and three booleans plus a procedure call for active event sensors.

Note that without changing the C compiler, it is not possible for BEE to know when a routine is exited. Thus in the case of E_PROCEDURE events, the return statements must be replaced by event sensors indicating the termination of the event range. BEE therefore provides two macros RET and RETURN for the termination of event ranges:

```
#define RET \
        if ((_eid_ > 0) && Event_on) { \
                Event_Terminate_Procedure(_eid_); \
                return; \
        } else \
                return

#define RETURN(v) \
        if ((_eid_ > 0) && Event_on) { \
                Event_Terminate_Procedure(_eid_); \
                return (v); \
        } else \
                return (v)
```

The END macro is syntactic sugar needed to close the compound statement opened by the BEGIN sensor:

```
#define END(name) \
        } \
     }
```

## 6.2 Performance Analysis of BEE

The performance analysis of any event processing system depends critically on the application and its instrumentation but there is currently no set of parallel benchmarks that cover a wide spectrum of applications. We characterize BEE's performance in terms of three parameters: the event rate, the runtime overhead experienced by the client and the event latency.

1. The *event rate* is the number of events that can be generated per time unit by an instrumented client program. The event rate depends on several parameters, including the event frequency, event density, event size and event interpreter service time. The performance of a runtime monitoring tool is also

influenced by additional parameters such as the cost of queueing X events for graphics based event interpreters.

2. The *client overhead* is the additional runtime needed to execute a client program instrumented for event processing compared with an uninstrumented program. More precisely, client overhead contains the execution time for the following activities: Encountering the event sensor, calling the event generator to assemble the event fields, calling the event handler to traverse the list of attached event interpreters and calling the `Every` function. If a local event interpreter is attached, the client overhead also measures the time to execute an empty `Every` function. In the case of a remote event interpreter, the client overhead includes the time of a nonblocking send of the event message. The client overhead depends mainly on the event rate, but it is of course a function of many other parameters as well. If remote event interpreters are attached, the client overhead is also a function of network parameters such as buffer size, retransmission rates and acknowledgement failures.

3. The *event latency* measures the time from when an event sensor is encountered to the point when it is interpreted by the event interpreter. Event latency is an important metric to characterize the real-time capability of the event system.

Given the lack of a good benchmark we have used the following event configuration: An instrumented client executing a loop that generates events at a constant rate of EGR event/sec and an event interpreter processing EIR events/sec with an event service time of EST per event. The execution time is then compared with the execution time of the same but uninstrumented client. The instrumented client generates events as fast as possible, and the event interpreter is "empty", that is, its `Every` function consists of an empty body.

The maximum event rate was computed by $N/T_{EI}$, where N is the number of generated events and $T_{EI}$ the measured execution time of the benchmark attached to an event interpreter. The client overhead was computed with the formula $(T_{EI} - T)/N$ where $T_{EI}$ is defined as above and T is the execution time of the benchmark with event processing turned off. The event latency was determined in three separate experiments, adding up the times obtained from each of these measurements:

1. The first experiment measures the time spent in the client program from the point when an event sensor is encountered. This is the same as the client overhead.

2. The second experiment measured the latency of a message between two host nodes across the network. We repeatedly sent an event message to another host which simply echoed it back to the sender. After measuring the round trip time for the message for a large number (1,000,000) we divided it by two to obtain the latency of the network.

3. In the third experiment we measured the time it took an event interpreter to receive an event, interpret it and be ready for the next incoming event. The event interpreter used in the experiment was the empty interpreter described above.

The client program generated 100,000 events for two different classes, the predefined event E_PROCEDURE (48 bytes on a Sun) and a user defined event (128 bytes). The event rate was controlled by changing the inter-event time via a program variable. The programs were compiled with an optimizing C compiler and the measurements were performed on the NECTAR prototype with one HUB using only Sun4/330 workstations as Nectar nodes. The numbers reported below should only be used when comparing BEE with other event processing systems.

For the empty client attached to a local (empty) event interpreter, the maximum event rate was 52,789 events/sec for E_PROCEDURE events and 28855 events/sec for the user defined event. The minimal client overhead was measured with 18 μsec for predefined PROCEDURE events and 34 μsec the user defined event.

For an empty client attached to an remote event interpreter the client overhead for the E_PROCEDURE event was 151 μsec. The reason for this low overhead is that all of the protocol processing for communication over the NECTAR network is done on the CAB communication processor, in parallel with the execution of the application on the host. The event latency was measured with 285 μsec/event. The maximum event rate between two Sun4/330 nodes was

measured with 6544 events/sec for E_PROCEDURE and 5733 events/sec for the user defined event. The event rate is limited by the VME bus that connects the host and the CAB [6].

The performance of BEE across heterogenous machines is influenced by the additional task of converting events into Ascii format before sending them and converting them back before interpreting them. If one has the choice, event interpreters should be placed on slower nodes. The event rate is higher if the sending host is faster than the receiving node: For example, for a Sun/Vax combination, the maximum event rate is 1070 events/sec if the client is on a Sun compared with 269 events/sec if the client runs on a Vax.

In the following we show the client overhead as a function of the event rate for various event configurations where the event rate is increased by the decreasing the time between events in the client. Figure 11 shows the client overhead of a single client on NECTAR connected to single event interpreters for different combinations of event rates and event service times. The event size in these measurements was 160 bytes. Each point in the graph represents a measurement. Each line connects all the measurements for a specific event interpreter. EI1, EI5 and EI8

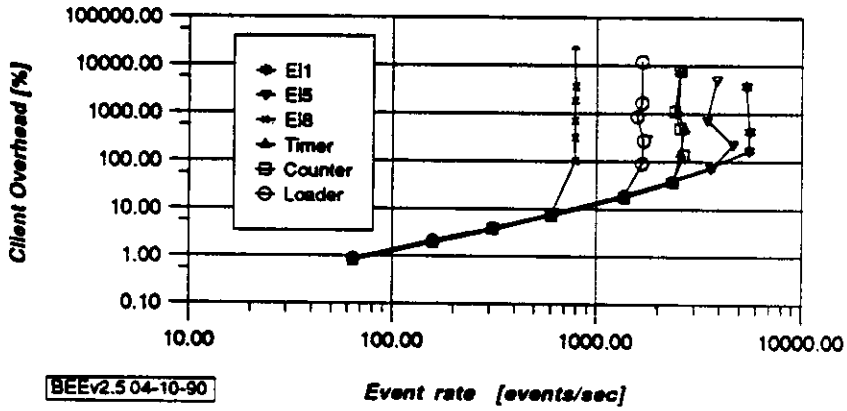


**Figure 11:** Bee's Client Overhead as a Function of the Event Rate (NECTAR)

represent event interpreters with event service times of 30 µsec, 100 µsec and 300 µsec, respectively. For EI1, the client can generate events at the maximum event rate. In the case of the event interpreters EI5 and EI8 we encounter saturation at lower event rates because of their higher event service times. When saturation is encountered some of the lines go "backward", that is, different client overheads seem to be caused by the same event rate. The reason for this is, of course, that the inter-event time and not the event rate is the true independent variable in the measurements. A smaller inter-event time increases the number of message retransmissions due to timeouts, which in turn increases the client overhead.

To summarize the results of figure 11, we can see that the client overhead of BEE is less than 1% for event rates up to 80 events/sec, and 10% for about 800 events/sec for 160 byte events. It depends on the application, whether these event rates are acceptable. If the client generates more than 1500 events/sec, BEE's performance degrades significantly. This includes the performance of BEE's default event interpreters time profiler (**Timer**), frequency counter (**Counter**) and load meter (**Loader**). Note that these measurements were taken from a client using no event

aggregration. The client overhead can be reduced further if the client program collects events before sending them as an aggregate to the event interpreter. Another possibility is the use of dormant event sensors which are polled by the event interpreter. If the event interpreter is X based, the overhead can also be reduced by increasing the screen update rate and update time of the event interpreter.

Figure 12 measures the performance of BEE configurations in which the client is connected to several event interpreters residing on different network nodes (For the experiment we used identical empty event interpreters). It

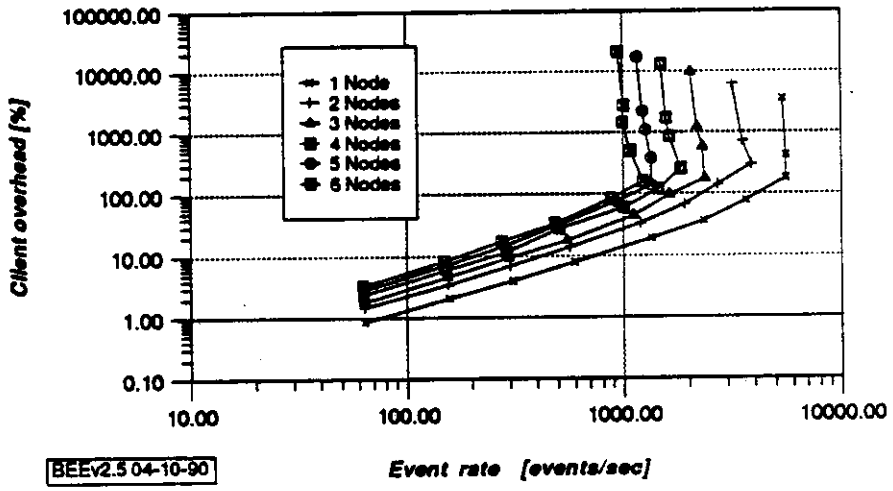


**Figure 12:** Client Overhead for Multiple View Configurations (NECTAR)

shows a slight increase in the client overhead when more event interpreters are connected. The reason is that, in the current version of BEE, a client sends a separate event message to each of the attached event interpreters. The use of broadcast or multicast messages might decrease the client overhead for large numbers of connected event interpreters, but this is currently an unresolved question.

The last experiment measured the performance of BEE configurations in which a single event interpreter is connected to several clients each of them sending event messages. Such configurations are useful for the implementation of distributed debuggers and load meters. Figure 13 shows that BEE's performance is largely independent of the number of attached clients for individual client event rates up to 1000 events/sec.

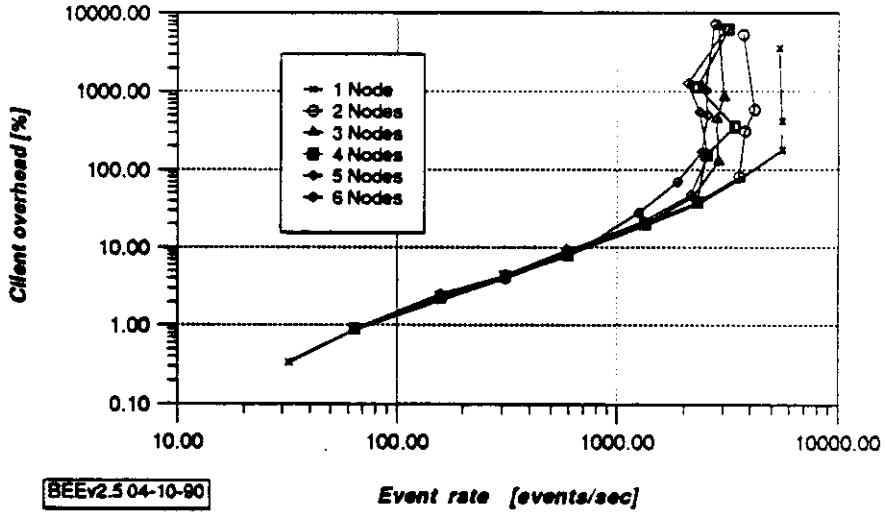


**Figure 13:** Client Overhead for Multiple Client Configurations (NECTAR)

# 7 Using Bee

BEE provides macros for the simple and fast instrumentation of clients with predefined event interpreters. The use of these macros for the C programmer is explained in Section 7.1. Section 7.2 describes the environment for executing instrumented client programs and event interpreters in under NECTAR and UNIX. Section 7.3 describes the views that can be associated with event interpreters and in Section 7.4 we discuss the default event interpreters provided with BEE. Section 7.5 explains how to write a customized event interpreter if none of the predefined event interpreters satisfies the needs of the user.

## 7.1 Instrumenting the Client

The interface to the event kernel is contained in the file **bee.h** which has to be included in the client program:

```
#include <bee.h>
```

To instrument routines written in C (see figure 14) with E_PROCEDURE events, the routine bodies must be enclosed with the BEGIN/END sensors described in section 8.2, page 30 replacing the usual curly brackets. If the routine is a procedure, a RET must be placed right before the END macro. If it is a function, each `return(value)` statement must be replaced by a `RETURN(value)` macro. Examples of instrumented client C routines are shown in figure 15.

```
void foo ()
    {
       /* uninstrumented procedure */
    }

int bar  ()
    {
       int n;
       /*
        * uninstrumented function
        */
       return(n);
    }
```

**Figure 14:** Uninstrumented Client Routines

```
void foo ()
    BEGIN(foo)
       /*
        * instrumented procedure
        */
       RET;
    END(foo)

int bar  ()
    BEGIN(bar)
       int n;
       /*
        * instrumented function
        */
       RETURN(n);
    END(bar)
```

**Figure 15:** Instrumented Client Routines

Inside the client, event interpreters are located with `Event_Lookup_Interpreter` functions (see Section 8.7). For example, to connect to the default frequency counter `"bee_counter"` and report the frequency of all routines whose names start with "foo", insert the following code fragment at appropriate places in the client

program:

```
#include <bee.h>

int Counter;

 Counter = Event_Lookup_Remote_Interpreter("bee_counter",
                                           E_PROCEDURE, "foo");
 /*
  *  Code section calling "foo*" routines
  */

Event_Detach_Interpreter(Counter);
```

The following example shows three things: 1) the definition of a user defined event E_SYSTEM of type `float` with a size of 4 bytes, 2) the generation of an event of the newly defined event class, and 3) the attachment to an event interpreter service **"bee_load"**:

```
long value;
Event_class_t E_SYSTEM;

E_SYSTEM = Event_Register_Class("E_SYSTEM", "%f", 4);

value = ...;
EVENT(sensor1, E_SYSTEM, E_POINT, &value);

EI = Event_Lookup_Remote_Interpreter("bee_load",E_SYSTEM,"");
```

If it is desirable to have all events interpreted by the event interpreter "service", the client code should contain the following statement:

```
Event_Lookup_Remote_Interpreter(<service>, E_ALL,"");
```

The following NECTAR example shows, how to get a frequency count for client procedures executed on Nectar node 0x5504:

```
int Counter;

 if (Nectar_id == 0x5504) {
    Counter = Event_Lookup_Remote_Interpreter("bee_counter",
                                              E_PROCEDURE,"");
   }
```

It is easy to see, how this example can be generalized to do selective tracing for certain tasks or processes.

## 7.2 Environment Variables

Instrumented client programs and event interpreters can only be executed if the global name server has been started. The environment variable BEE_NS_HOST must contain the location of the name server.

BEE's X based event interpreters read the location of the display from the environment variable DISPLAY unless it is passed as a parameter to the command line (see section 7.5).

The environment variable BEE_ARCH has to contain the architecture of the machine on which the client or event interpreter is running, which must be one of the following values: `sun`, `vax` or `cray`.

If the environment variable BEE_REPLAY is set to the name of an event trace file, the event interpreter reads events from this file instead of waiting for event messages. BEE's replay facility is quite useful when developing a new

event interpreter, because it can tested independent from the communication system.

Clients and event interpreters can be started as regular Unix programs. On NECTAR, the `start_appl` facility [4] allows the user to start event configurations without having to know the location or the command interface of default event interpreters.

## 7.3 Bee Views

The interpretation of the incoming event stream is called a BEE *view* and we distinguish textual and graphical views. A *textual* BEE view presents the events in a typescript window or file and incoming events are printed textually. The typescript view is useful for small monitoring tasks, but it is not very helpful for monitoring the performance of network programs. Even if the event streams are condensed it is often impossible for the user to understand the behavior of the client from textual information. One of the reason is that the partial event streams coming from each of the nodes are not separated.

Graphical views condense the incoming events and allow to comprehend much more complex situations. They are built on top of the X library and impose a higher event latency than textual BEE views. A graphical view reacts to client events triggered by event sensors as well as to X events.

A graphical view creates a graphical area inside an X window (or widget) when the event interpreter receives an `Init` event messages from the client. Hints for the X window, such as position, width and height can be provided by the client with the event kernel function `Event_Window()`. After its creation, the view is updated either as the result of a BEE event message or an X event. The update frequency for `Every` event messages is controlled by the event kernel functions `Event_Update_Rate()` and `Event_Update_Time()`. A BEE view is also updated when the user resizes or uncovers the X window associated with the view. Finally, a BEE view is always updated when the event interpreter receives a `Final` event from a client.

BEE provides three predefined graphical views: histogram, piechart, and linegraph. In the *histogram* view, the output is presented in a coordinate system where horizontal columns starting at the y axis represent event sensors encountered in the client program[5]. The current event sensor values are shown along the x axis. An event sensor is displayed only if it is actually encountered during the execution. Layout hints for the coordinate system associated with the histogram view can be set with the event kernel function `Event_Histogram_View()` described in Section 8.9, page 45. If a sensor value exceeds the current maximum value on the x axis, the histogram is redrawn with a larger x axis limit and the event sensor columns are automatically resized.

The histogram view is very useful for visualizing frequency counts or time profiles. Figure 16 shows a histogram based frequency counter taking events from a client which executed the following program fragment before connecting to the event interpreter:

```
axis_type_t  x_axis_type = LINEAR;
axis_type_t  y_axis_type = LINEAR_NO_NUMBERS;
long         x_limit = 1000000 , y_limit = 0;

 Event_Window("Counter", 765, 573, 375, 320, BLUE, STEELBLUE);
 Event_Histogram_View( "Frequency Coun", " Calls",  "",
                       x_axis_type,  y_axis_type, x_limit, y_limit);
```

[4] For a description of `start_appl` we refer to the man page start_appl(1).

[5] The histogram view is horizontal so that event sensor names can be read easily.

The time stamp of the latest event received by the event interpreter is called **EI Time**. It is the client's timestamp corrected by the event interpreter offset determined at startup time (see Section 1) and is usually shown at the top of the window below the title bar.

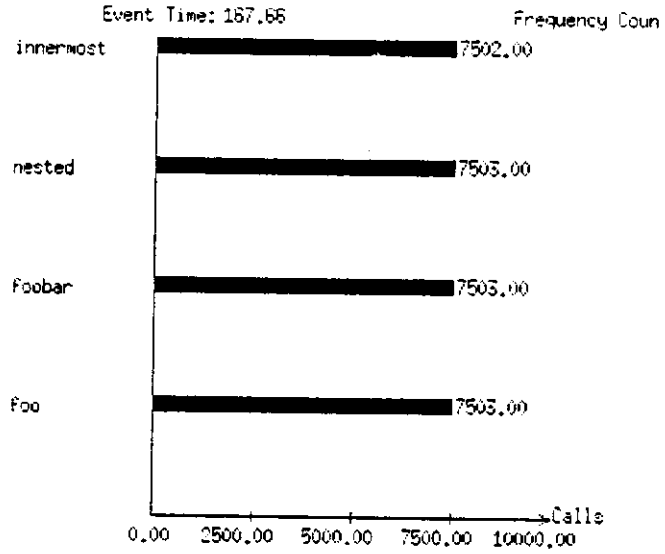


**Figure 16:** Histogram view of a Frequency Counter

A *piechart* view is presented as a sectored circle in the X window, where each sector represents an event sensor. The size of the sector represents the current value of the sensor. If a new sensor is added, all sectors in the circle are recomputed and resized. Sectors are shown only for sensors actually encountered during the execution. The sum of all displayed sensor values always add up to 100%. Event sensor values below a certain threshold are not displayed, but are lumped together in a black painted sector. Figure 17 shows the piechart view of a time profiler taking events from the same client used in the previous figure.

The *linegraph* view presents event sensor values as functions over time. The client has many choices for the layout of a linegraph which can be set with the event kernel function `Event_Linegraph_View()` described in Section 8.9, 46. The y axis displays either absolute sensor values or their percentage in relation to the total sum of event sensor values. Percentages are computationally cheaper than sensor values, because in the former case the y axis never needs to be rescaled. It is also possible to specify how the curves are shown in relation to each other. In the *stacked mode*, each sensor line serves as the x axis of the next sensor line, with the exception of the first sensor which uses the x axis of the coordinate system as base line. In the *unstacked mode*, all event sensor values are drawn with respect to the x axis of the coordinate system.

The linegraph view is useful for the visualization of load balancing algorithms. Figure 18 shows a load meter taking events from a client instrumented with predefined and user defined event sensors using a stacked linegraph view. Before connecting to the event interpreter, the client issued the event kernel call:

```
axis_type_t  x_axis_type = LINEAR;
axis_type_t  y_axis_type = PERCENT;
Event_Linegraph_View("Load Meter", "secs","",
                     x_axis_type,  y_axis_type,
                     1000000,100,20,TRUE);
```

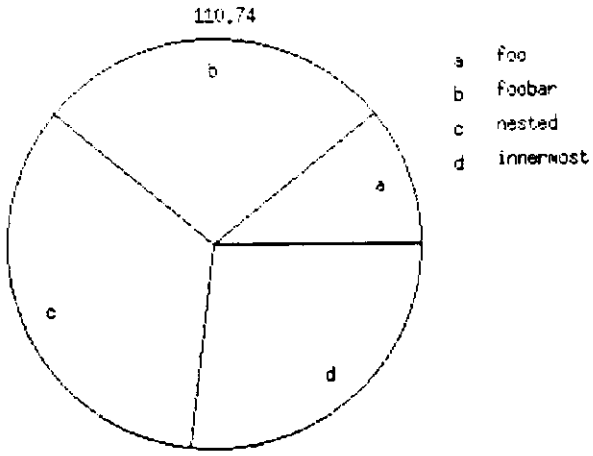


**Figure 17:** Piechart view of a Time Profiler

Note that, in addition to the event sensors of type E_PROCEDURE displayed in the previous figures, figure 18 also shows the values of user defined event sensors called **SensorA, SensorB, SensorC** and **pinot**[6]. The current sensor values are represented in a menu shown at the right of the coordinate system. The correspondence between lines and sensor values is positional: The lowest menu entry belongs to the lowest function in the coordinate system. If sensor lines are too close to be distinguishable, the user can resize the X window and the linegraph is redrawn in the larger window.

## 7.4 Bee's Default Event Interpreters

BEE provides a small set of predefined event interpreters which are described in this section. Default event interpreters are quite flexible monitoring tools, because the definition of their service is user definable, and they can be started up with any of the three views described in Section 7.3.

### 7.4.1 Frequency Counter

BEE's frequency counter counts the number of routine calls in a client program. The frequency counter can be executed locally or remotely. The local event interpreter functions are called `Bee_Counter_Init()`, `Bee_Counter_Every ()` and `Bee_Counter_Final()`, respectively. The name of the remote service is `"bee_counter"`.

The frequency counter is available for all the views described in section 7.3.

The `typescript view` frequency counter prints a summary of the execution profile upon the execution of the client program or when the event interpreter is detached. An example of an execution profile obtained with a local frequency counter is shown in figure 19:

---

[6]In the linegraph view, the lines may accumulate to more than 100% because of rounding errors during the conversion of percentages of type **float** into values of type **int**.

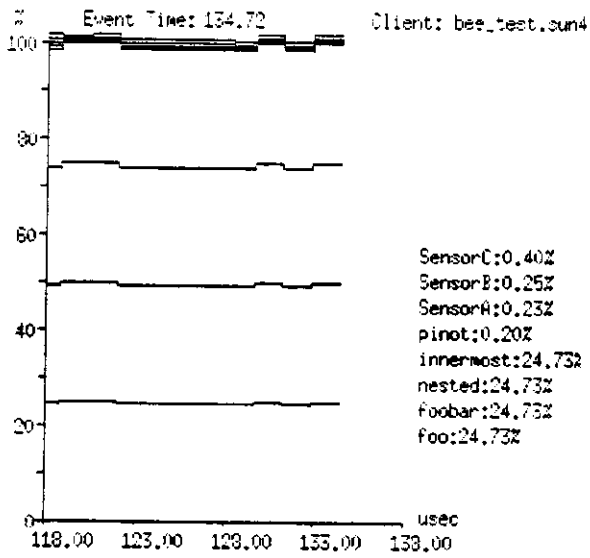


**Figure 18:** Linegraph view of a Load Meter

```
[main] Name                                    Frequency
[main] ----------------------------------------
[main] foo                                          2020
[main] foobar                                       2000
[main] nested                                       2000
[main] innermost                                    2000
[main] bar                                            10
[main]
```

**Figure 19:** Execution Summary of a Frequency Counter

### 7.4.2 Time Profiler

BEE's time profiler measures the time spent in event ranges of the instrumented client program. The time profiler can be executed locally or remotely. The local event interpreter functions are called `Bee_Timer_Init()`, `Bee_Timer_Every ()` and `Bee_Timer_Final()`, respectively. The name of the remote service is `"bee_timer"`.

If event sensors are nested, the time reported for the inner event range is not counted in the outer event range. For example, in the code fragment:

```
EVENT(T1, E_PROCEDURE, E_ACT, 0)
<Time_A>
EVENT(T2, E_PROCEDURE, E_ACT, 0)
<Time_B>
EVENT(T2, E_PROCEDURE, E_TERM, 0)
<Time_C>
EVENT(T1, E_PROCEDURE, E_TERM, 0)
```

the timing distribution will be reported as follows:

```
T1                          Time_A + Time_C
T2                          Time_B
```

The `typescript view` time profiler prints a summary of the execution profile upon the execution of the client

program or when the event interpreter is detached. For each encountered sensor range it prints the average time spent in the range, its variance, the total time and the percentage. An example of an execution profile obtained with a remote time profiler counter is shown in figure 20:

```
[bee_timer] Name            Mean       Variance      Cum.Time     Percent
[bee_timer] ------------------------------------------------------------
[bee_timer] innermost    0.000173   0.000000      0.001735        0.02
[bee_timer] nested       0.002934   0.000009      5.867842       75.06
[bee_timer] foobar       0.000347   0.000000      0.694601        8.88
[bee_timer] foo          0.000452   0.000000      0.903031       11.55
[bee_timer] bar          0.000174   0.000000      0.350514        4.48
[bee_timer]                                     ----------------------
[bee_timer]                                     7.817723      100.00
```

**Figure 20:** Execution Summary of a Time Profiler

When the `Final()` function is executed, the time profile summary is appended to a file `~/<client_program_name>.times` in the user's home directory independent of the viewing mode.

### 7.4.3 Load Meter

BEE's load meter reports the load of event sensors in an instrumented client at a user defined rate or time and is available only in a X based version. The name of the default service is `"bee_load"`.

The histogram view of the load meter displays the latest values for each event sensor. The linegraph and piechart view report the load as follows: First they compute the sum of the latest values of each sensor and then they display each sensor value as a percentage of the total sum. Note that this is only an approximation of the actual load, especially if some of the clients do not send their events as regularly as others. This can happen, for example, when a client is swapped out by the operating system right before sending its event.

### 7.4.4 Event Filer

BEE's event filer records all incoming event messages in an event file. The event file can be then used for replay or postmortem monitoring and for testing and debugging of customized event interpreters (see Section 7.5). The name of the remote service is `"bee_filer"`. The event file format is described in Section 9.10.

### 7.4.5 Remote Printer

BEE provides a remote printing service `"bee_print_msg"`. If the event kernel function `Event_Remote_Print()` has enabled remote printing, the output of PRINT_MSG macros inserted in the client program is sent to the event interpreter instead of being printed in the client's typescript window. The service can collect and/or filter prints statements originating in any of the network nodes. Its main application is for debugging of network programs when higher level tools are not available.

### 7.4.6 Tracer

BEE's tracer traces the entry and exit of event ranges in instrumented client program. The tracer can be executed locally or remotely. The local event interpreter functions are called `Bee_Tracer_Init()`, `Bee_Tracer_Every()` and `Bee_Tracer_Final()`, respectively. The name of the remote service is `"bee_tracer"`.

## 7.5 Customized Event Interpreters

Bee is extensible. If the none of the predefined event interpreters can aid in monitoring the computation as desired by the user, it is possible to write a customized event interpreter. In this case, the user has to provide the definition of the three functions `Init()`, `Every()`, `Final()` and a call to `Event_Enter_Interpreter()` which enters the event interpreter with the global name server.[7].

Figure 21 shows the definition of an event interpreter called **MY_EI** providing the service "custom_service". When connected to a client, it prints out "Hello world" whenever an event occurs.

```
#include <bee.h>

Event_return_t MY_EI_Init(E)
Event_t E;
{
  return(E_SUCCESS);
}

Event_return_t MY_EI_Every(E)
    Event_t E;
{
  printf("Hello world\n");
  return(E_SUCCESS);
}

Event_return_t MY_EI_Final(E)
Event_t E;
{
  return(E_SUCCESS);
}

int MY_EI (E_Class, Filter)
Event_class_t E_Class; char * Filter;
{
  Event_Enter_Interpreter("custom_service",
                MY_EI_Init, MY_EI_Every, MY_EI_Final,
                E_Class, Filter,
                E_REPLACE, E_KEEP_SENSORS);
);
}
```

**Figure 21:** A user defined event interpreter MY_EI

To attach the event interpreter to a client and print out the string only for procedures whose name start with "foo", we add the following code fragment to the client:

```
#include <bee.h>

int My_Event_Interpreter;

My_Event_Interpreter = MY_EI(E_PROCEDURE, "foo");
```

**Figure 22:** Attaching MY_EI with filter "foo" to a client

The user interface of a customized event interpreter should follow the interface of default event interpreters which

[7]To ease the programming job, a template for the definition of an empty event interpreter is available in the sub-directory **$BEE_DIR/bee_template.c**. It is also a good idea to look at the existing default interpreters and see if it is possible to modify them to get the desired functionality.

expect three command line arguments: **View**, **Service** and **Display**. The first argument **View** specifies how the event information is to be displayed and can be `typescript, hist, pie or line`. The second argument **Service** is the name of the service by which the event interpreter is known in the network. It is entered into the BEE domain of the name server with `Event_Enter_Service()` (which is called by `Event_Enter_Interpreter()`). The same event interpreter can be started with different service names. This permits multiple instances of event interpreters listening to different clients. For example, a load meter could be instantiated twice, once as a work meter with service name "work" showing the work performed on various nodes as well as a throughput meter with service name "throughput" showing the current communication bandwidth used by client programs. The optional third argument **Display** is X specific and used by the event interpreter to display of the results. **Display** must be specified in the well known X format **host:server:screen**.

## 8 Functional Specification

The event kernel is a set of functions to instrument client programs and build event interpreters. In this section we first give an overview of the event kernel functionality, followed by a detailed discussion of each its functions[8].

BEE provides **event sensors** for the predefined event classes E_EVENT and E_PROCEDURE as well as a general event sensor for user defined event classes. An event sensor can be in one of four states: uninitialized, active, passive or dormant. An uninitialized sensor is made active by calling `Event_Register_Name()` when the sensor is encountered the first time during the execution. An active sensor is passed to the event generator for further processing whenever the sensor is encountered. A passive sensor is not passed to the event generator. A dormant sensor does not generate events; its value must be queried by an event interpreter request.

**Event sensor functions** can be used to control the client overhead caused by event sensors. For example, the function `Event_Sensor_Filter()` changes a set of active sensors into passive sensors and vice versa.

**Event Initialization functions** are called at the startup and finish of a client or thread. The functions `Event_Initialize()` and `Event_Finalize()` are called at the beginning and end of the execution. The functions `Event_Create_Table()` and `Event_Cleanup_Table()` are called at the creation and deletion of threads.

The **Event naming functions** define new names of event sensors and event classes. At initialization time, BEE enters the predefined event classes into the event table in `Event_Initialize()` using these functions. They can also be called by users to define new event classes.

The **Event generator functions** assemble the component of an event and hands it over to the **Event handler functions** which in turn dispatch the events to associated event interpreters. Handler functions also provide the enabling/disabling of event classes. **Event interpreter functions** permit the disabling/enabling of event interpreters.

**Event service functions** make event interpreters and clients known to the name server. The communication between client and event interpreters using service functions is via BEE ports. In NECTAR, a BEE port is equivalent to a Nectarine port [20] and in the UNIX implementation it is a socket. Each client server is accessible by its client server port. An event interpreter issuing remote event kernel commands has to connect to the client server first. This is done by calling `Event_Lookup_Service()` to obtain the client server port followed by a call to `Event_Attach_Client_Port()`.

---

[8]For pragmatic reasons we have chosen use C as specification language. Because C does not have a package concept, each event kernel function is prefixed with the string `Event_`. Event kernel constants are written in upper case and prefixed with `E_`.

**Event interpreter control functions** give the client the possibility to provide the event interpreter with hints concerning screen update rate, layout and window information.

A major part of the event kernel is the event protocol, governing the exchange of information between clients and event interpreters. The **Event protocol functions** provide the functional interface of the event protocol and are described in Section 9.

## 8.1 Event Sensors

BEE provides invasive event sensors for the instrumentation of client programs. For efficiency reasons they are implemented using features such as inline expansion and variables with static extent. Because not all programming languages provide these features, BEE offers a collection of event sensors from which the programmer has to select the appropriate ones. Section 8.1.1 describes sensors that can be used for the instrumentation of program written in any language. Section 8.1.2 describes sensors that should be used for the instrumentation of C programs.

### 8.1.1 Language Independent Sensors

**Event_Sensor**

An event sensor for a user defined event class (with EID parameter).

INTERFACE:

```
Event_Sensor(Name, Class, Attr, Variant, EID)
char * Name;
Event_class_t Class;
Event_attribute_t Attr;
pointer_t Variant;
int * EID;
```

PARAMETERS:

| | |
|---|---|
| **Name** | Event sensor name. |
| **Class** | Event class. |
| **Attr** | Event attribute. |
| **Variant** | Pointer to event data field. |
| **EID** | Event id. |

RETURNS: Nothing.

NOTES: For user defined events, the size and layout of `Variant` must be entered by `Event_Register_Class()`. The variant fields have to be filled by the client program. The value of **EID** is set the first time the event sensor is called and it must not be changed by the client program.

### 8.1.2 C Language Sensors

C provides static variables (similar to ALGOL's **own**) which can be initialized at declaration time. This feature is used in the definition of the sensors `EVENT`, `BEGIN`, `END`, `BEGIN_EVENT`, `END_EVENT`, and `POINT_EVENT` hiding the existence of the EID. This results in a slightly more efficient sensor implementation compared with sensors that require the EID as parameter.

**EVENT_SENSOR**

An event sensor for a user defined event class (with EID parameter).

INTERFACE:

```
EVENT_SENSOR(Name, Class, Attr, Variant, EID)
char * Name;
Event_class_t Class;
Event_attribute_t Attr;
pointer_t Variant;
int * EID;
```

PARAMETERS:

| | |
|---|---|
| **Name** | Event sensor name. |
| **Class** | Event class. |
| **Attr** | Event attribute. |
| **Variant** | Pointer to event data field. |
| **EID** | Event id. |

RETURNS: Nothing.

NOTES: For user defined events, the size and layout of `Variant` must be entered by `Event_Register_Class()`. The variant fields have to be filled by the client program. `EID` must be declared in the client program; its value is set the first time the event sensor is called and it must not be changed by the client program.

### EVENT

An event sensor for a user defined event class.

INTERFACE:

```
EVENT(Name, Class, Attr, Variant)
char * Name;
Event_class_t Class;
Event_attribute_t Attr;
pointer_t Variant;
```

PARAMETERS:

| | |
|---|---|
| **Name** | Event sensor name. |
| Class | Event class. |
| Attr | Event attribute. |
| **Variant** | Pointer to event data field. |

RETURNS: Nothing.

NOTES: EVENT is a macro that hides the existence of the EID from the user. For user defined events, the size and layout of `Variant` must be entered by `Event_Register_Class()`. The variant fields have to be filled by the client program.

### BEGIN_PROCEDURE

An event sensor for the activation of an event range of class E_PROCEDURE.

INTERFACE:

```
BEGIN_PROCEDURE(Sensor_Name)
char * Sensor_Name;
```

PARAMETERS:

| | |
|---|---|
| **Sensor_Name** | Event sensor name. |

RETURNS: Nothing.

NOTES: The macro `BEGIN` can be used instead of `BEGIN_PROCEDURE`, but in this case a string constant is expected as parameter.

### END_PROCEDURE

An event sensor for the termination of an event range of class E_PROCEDURE.

INTERFACE:

```
END_PROCEDURE(Sensor_Name)
char * Sensor_Name;
```

PARAMETERS:

**Sensor_Name** Event sensor name.

RETURNS: Nothing.

NOTES: The macro END can be used instead of END_PROCEDURE, but in this case a string constant is expected as parameter.

### BEGIN_EVENT

An event sensor for the activation of an event range of class E_EVENT.

INTERFACE:

```
BEGIN_EVENT(Sensor_Name)
char * Sensor_Name;
```

PARAMETERS:

**Sensor_Name** Event sensor name.

RETURNS: Nothing.

### END_EVENT

An event sensor for the termination of an event range of class E_EVENT.

INTERFACE:

```
END_EVENT(Sensor_Name)
char * Sensor_Name;
```

PARAMETERS:

**Sensor_Name** Event sensor name.

RETURNS: Nothing.

### POINT_EVENT

Generate the an event point of event class E_EVENT.

INTERFACE:

```
POINT_EVENT(Sensor_Name)
char * Sensor_Name;
```

PARAMETERS:

**Sensor_Name** Event sensor name.

RETURNS: Nothing.

## 8.2 Event Sensor Functions

### Event_Sensor_Max

Maximum number of event sensors allowed in client program.

INTERFACE:

```
Event_Sensor_Max()
```

PARAMETERS: none:

RETURNS: Maximum number of event sensors.

**Event_Sensor_Control**

Disable/enable all event sensors of a given event class.

INTERFACE:

```
Event_Sensor_Control(Class, Flag)
event_class_t Class;
boolean_t Flag;
```

PARAMETERS:

| | |
|---|---|
| **Class** | Event class. |
| **Flag** | If TRUE, turn on all event sensors for event class **Class**, otherwise turn off all event sensors for event class **Class**. |

RETURNS: Nothing.

NOTES: **Event_Sensor_Control(<class>, FALSE)** for all predefined event classes `<class>` is called by **Event_Initialize()**.

**Event_Sensor_Filter**

A filter to be applied to all event sensors of an event class.

INTERFACE:

```
 Event_return_t Event_Sensor_Filter(Class, F)
Event_class_t Class;
char * F;
```

PARAMETERS:

| | |
|---|---|
| **Class** | Event class to which the filter applies. |
| **F** | Value of the new global sensor filter. All currently registered event sensor names are compared with **F** and if **F** is not a substring of the event sensor name, the sensor is made passive. The initial value of the global sensor filter is ("?"). |

RETURNS: Nothing.

NOTES: Currently BEE supports only one event sensor filter for all event classes, that is, the event class parameter is ignored. Any uninitialized sensor encountered after the installation of the new filter, whose name is not a substring of the global sensor filter, is also made passive.

## 8.3 Event Initialization Functions

**Event_Initialize**

Initialize the event processing facility for client.

INTERFACE:

```
Event_return_t Event_Initialize (Client)
char * name_of_client;
```

PARAMETERS:

| | |
|---|---|
| **Client** | Pathname of the client. |

RETURNS:

NOTES: Under NECTAR, `Event_Initialize()` is automatically called by the function `Nectar_Init()`. Under UNIX `Event_Intialize()` is called by `Unix_Init()`. Both of these functions are not part of BEE.

**Event_Finalize**

Wrap up event related activities in client program.

INTERFACE:

```
Event_return_t Event_Finalize (Name_of_client)
char * Name_of_client;
```

PARAMETERS:

**Name_of_client** Pathname of the client.

RETURNS: E_SUCCESS if successful, E_FAILURE otherwise.

NOTES: `Event_Finalize()` scans the event table(s) associated with the client and executes the final action for all event interpreters attached to enabled event classes. Local event interpreters are called to execute their `Final()` function, remote event interpreters receive an E_FINAL message. Remote event interpreters send an acknowledgement E_KILL back to the client after receiving the final message to indicate that they do not need to communicate with the event client server any longer. After receiving the acknowledgements from all attached event interpreters, `Event_Finalize()` broadcasts a CLIENT_DEATH messages, frees the memory allocated for the event tables, kills the event client server and exits the client. This protocol ensures that the client is alive for post mortem requests by event interpreters.

**Event_Create_Table**

Allocate an event table (system or task event table).

INTERFACE:

```
Event_return_t Event_Create_Table (ET, ENT)
pointer_t ET;
pointer_t ENT;
```

PARAMETERS:

**ET** Pointer to event table.

**ENT** Pointer to event nesting table.

RETURNS: E_SUCCESS if successful, E_FAILURE otherwise.

NOTES: This function is always called by `Event_Initialize()` to allocate the system event table and system event nesting table. The event nesting table keeps track of event sensor ranges that have been activated but not yet terminated. Event nesting tables are useful for debugging of the client instrumentation event ranges and they are needed by the the time profiler to compute the times spent in an event range. If event table inheritance is enabled (see Section 8.10), this function is also called whenever a task is created. In NECTAR, the two arguments **ET** and **ENT** point to fields in the Nectarine task control block. A task event table inherits all event classes from the system event table, but it does not inherit the associated event interpreters.

**Event_Cleanup_Table**

Deallocate an event table (system or task event table).

INTERFACE:

```
Event_return_t Event_Cleanup_Table (ET, ENT)
Event_table_t ET;
Event_nest_table_t ENT;
```

PARAMETERS:

**ET** Pointer to event table.

**ENT** Pointer to event nesting table.

RETURNS: Nothing.

NOTES: Deletes the event table and event nesting table of a task. In Nectar, this function is automatically

called by `N_Kill_Self()`, whenever a Nectarine task is finished. `Event_Cleanup_Table()` is also be called by `Event_Finalize()` when the client finishes execution on a node. In this case it deallocates the system event table.

## 8.4 Event Naming Functions

**Event_Client_Name**

Return name of client program.

INTERFACE:

```
char * Event_Client_Name()
```

PARAMETERS: None.

RETURNS: The name of the attached client.

NOTES: If the process is attached to a remote client, request the name from the client server, otherwise do a local lookup.

**Event_Class_Name**

Return name of event class.

INTERFACE:

```
char * Event_Class_Name(Class)
Event_class_t Class;
```

PARAMETERS:

**Class** Event class descriptor.

RETURNS: The name of a predefined or user defined event class if it exists, "" if none exists.

NOTES: If the process is attached to a remote client, request the name from the client server, otherwise do a local lookup.

**Event_Class_ID**

Return the event class descriptor of a predefined or user event class.

INTERFACE:

```
Event_class_t Event_Class_ID(Name)
char * Name;
```

PARAMETERS:

**Name** Name of event class.

RETURNS: The event table index of the event class or -1 if **Name** is not known.

NOTES: If the process is attached to a remote client, request the class from the client server, otherwise do a local lookup.

**Event_Register_Class**

Define a new event class **Name** with a variant part of **Size** bytes.

INTERFACE:

```
Event_class_t Event_Register_Class(Name, Type, Size)
char * Name;
char * Type;
int Size;
```

PARAMETERS:

**Name** Name of event class.

**Type** Type descriptor of event variant. A type descriptor is has the same syntax as

the control string used by `printf()` and `scanf()` in C's I/O facility. Type contains the conversion specifications for each of the fields in the event variant and no other information.

**Size** Size of event variant. Depending on the compiler's allocation strategy, the size of the event variant might be larger than than the sum of the sizes of the fields of the event variant.

RETURNS: Event class descriptor.

NOTES: 1) The event variant descriptor is sent to a remote event interpreter as event message of type E_DESCRIPTOR immediately after the attacment of the client, if the event interpreter interprets this event class. 2) The following code fragments shows the definition of a user defined event E_SYSTEM of type integer with a data area of 4 bytes and attachment to an event interpreter **"bee_load"**:

```
Event_class_t E_SYSTEM;

E_SYSTEM = Event_Register_Class("E_SYSTEM", "%d", 4);

EI = Event_Lookup_Remote_Interpreter("bee_load", E_SYSTEM, "");
```

To generate an event of the newly defined event class, use the event sensor EVENT:

```
int value;
...
EVENT(sensor_name, E_SYSTEM, E_POINT, &value);
```

**Event_Register_Name**

Enter name of event sensor into BEE's internal tables and return its event sensor descriptor.

INTERFACE:

```
Event_id_t Event_Register_Name(Name, E_id)
char * Name;
Event_id_t * E_id;
```

PARAMETERS:

**Name** Name of event sensor.

**E_id** Client process unique event sensor id assigned by event kernel.

RETURNS:

NOTES: If the event sensor is active, `Event_Register_Name` calls `Event_Generate`.

**Event_Sensor_Name**

Return name of event sensor given an event sensor descriptor.

INTERFACE:

```
char * Event_Sensor_Name(E_id)
int E_id;
```

PARAMETERS:

**E_id** Client process unique event sensor id.

RETURNS:

## 8.5 Event Generator Functions

**Event_Generate**

Event_Generate is called by an active event sensor.

INTERFACE:

```
void Event_Generate(Class, Attr, EID, Variant)
     Event_class_t class;
     Event_attribute_t attr;
     Event_id_t EID;
     pointer_t variant;
```

PARAMETERS:

| | |
|---|---|
| **Class** | Name of Event class. |
| **Attr** | Attribute (see section 9.2. 51). |
| **EID** | Event id. |
| **Variant** | Pointer to event variant. |

RETURNS: Nothing.

NOTES: `Event_Generate` selects the appropriate event table (system or task event table), collects the components of the event and calls the event handler (see `Event_Handle`). Event processing is disabled during the generation of an event.

**Event_Regenerate**

Event_Regenerate passes an event message to the event handler without modifying the timestamp.

INTERFACE:

```
void Event_Regenerate (IN E)
     Event_t E;
```

PARAMETERS:

| | |
|---|---|
| **E** | Event message. |

RETURNS: Nothing.

NOTES: `Event_Regenerate()` is useful for multiple view monitoring of systems where a high communication bandwidth is not available. A single remote event interpreter receives event messages from one or more clients, filters them and passes them on to the attached (local) event interpreters.

## 8.6 Event Handler Functions

**Event_Handle**

Dispatch the event to all attached event interpreters.

INTERFACE:

```
Event_return_t Event_Handle (IN E)
    Event_t E;
```

PARAMETERS:

| | |
|---|---|
| **E** | Event (assembled by event generator). |

RETURNS: E_SUCCESS if successful, E_FAILURE otherwise.

NOTES: `Event_Handle` is called by `Event_Generate` only if the event class for the generated event is enabled and if at least one event interpreter is attached. If the event table is active and if the event class is enabled, the event message of type `E_EVERY` is dispatched to all event interpreters registered for this event class.

**Event_Handle_Rate**

Event handler dispatch rate.

INTERFACE:

```
Event_return_t Event_Handle_Rate(Class,
                                 NrOfEvents,
                                 Accumulation_Mode,
                                 Dispatch_Mode)
Event_class_t Class;
long NrOfEvents;
Event_accumulation_t Accumulation_Mode;
Event_dispatch_t Dispatch_Mode;
```

PARAMETERS:

**Class** Event class.

**NrOfEvents** Dispatch the (aggregate) event every **NrOfEvents** events.

**Accumulation_Mode**
Specifies how to accumulate events if event handler does not immediately dispatch events.

E_LATEST: Store only latest event.

E_ADD: Add new event value to aggregate event.

E_MEAN: Compute mean of all event values stored in event handler.

**Dispatch_Mode** A threshold for the event handler that specifies when to dispatch events. Possible values are:

E_HANDLE_IMM: Immediately dispatch encountered event.

E_HANDLE_BY_TIME:
Dispatch event aggregate every N secs.

E_HANDLE_BY_EVENT:
Dispatch event aggregate every N events.

RETURNS: E_SUCCESS if successful, E_FAILURE otherwise.

NOTES: `Event_Handle_Rate` allows the client program to influence the event client overhead by bundling events in the event handler before they are sent to the attached event interpreters. *Default value*: When a new event class `C` is entered into the event table, the event kernel calls `Event_Handle_Rate(C, E_LATEST, E_HANDLE_IMM)`.

**Event_Enable**

Enable the event class in the current event table.

INTERFACE:

```
Event_return_t Event_Enable (IN Class)
Event_class_t Class;
```

PARAMETERS:

**Class** Event class.

RETURNS: , E_SUCCESS if succesful, E_FAILURE if Class is unknown. If the call fails and Event_Verbose() is on, an error message is printed.

**Event_Delete**

Delete a user defined event class from BEE's internal tables.

INTERFACE:

```
Event_return_t Event_Delete(IN Class)
    Event_class_t Class;
```

PARAMETERS:

**Class** Event class.

RETURNS: E_SUCCESS if successful, E_FAILURE otherwise.

**Event_Disable**

Disables an event class in the event table.

INTERFACE:

```
Event_return_t Event_Disable (IN Class)
Event_class_t Class;
```

PARAMETERS:

**Class** Event class.

RETURNS: E_SUCCESS if successful, E_FAILURE otherwise.

## 8.7 Event Interpreter Functions

**Event_Enter_Interpreter**

Enter event interpreter into name server domain.

INTERFACE:

```
Event_Enter_Interpreter(Name,
                        Init,Every,Final,
                        Class, Filter,
                        Enter_Mode, Client_Death_Mode)
char * Name;
Event_return_t (* Init) ();
Event_return_t (* Every) ();
Event_return_t (* Final) ();
Event_class_t Class;
char * Filter;
Event_lookup_mode_t Enter_Mode;
Event_client_death_mode_t Client_Death_Mode;
```

PARAMETERS:

**Name** Pathname of the executable of the client program.

**Init** Pointer to the init function which must be defined as follows:

```
Event_return_t Init(Event)
Event_t Event;
```

**Every** Pointer to the every function which must be defined as follows:

```
Event_return_t Every(Event);
Event_t Event;
```

**Final** Name of the final function which must be defined as follows:

```
Event_return_t Final(Event)
 Event_t Event;
```

**Class** The event class to be interpreted.

**Filter** A string used for filtering events at event handling time. If the event name contains `Filter` as an initial substring, the event is handled, otherwise it is not handled.

**Enter_Mode** What to do if the service already exists in the name server (The enumerated type `Event_lookup_mode_t` is described in Section 8.8, page 42).

**Client_Death_Mode**

What to do with client sensors when the client dies:

E_DELETE_SENSORS Delete all information about client in event sensor array.

E_KEEP_SENSORS Keep event sensor information about client.

RETURNS: If the call is successful, there is no return. The message

```
"Event interpreter ready."
```

appears on the output and the EI server waits in an endless loop for events from registered clients or event interpreters. If the message

```
"Event interpreter not ready."
```

appears, the EI server could not be started up, usually for one of the following reasons:

- Another event interpreter with the same name was already entered in the BEE domain.
- The name server timed out.

NOTES: The pair <Name, EI server port> is entered into the name server domain BEE and the interpreter is started as a server waiting for messages from clients. Clients connect to the service with `Event_Lookup_Interpreter()`.

**Event_Lookup_Interpreter**

Look up event interpreter.

INTERFACE:

```
int Event_Lookup_Interpreter(Service,
                             Init, Every, Final,
                             Class, Filter, Location)
char * Service;
Event_return_t (* Init) ();
Event_return_t (* Every) ();
Event_return_t (* Final) ();
Event_class_t Class;
char * Filter;
EI_location_t Location;
```

PARAMETERS:

**Service** Service name of event interpreter.

**Init** Pointer to the init function which must be defined as follows:

```
Event_return_t Init(E)
Event_t E;
```

**Every** Pointer to the every function which must be defined as follows:

```
Event_return_t Every(E);
Event_t E;
```

**Final** Name of the final function which must be defined as follows:

```
Event_return_t Final(E)
Event_t E;
```

**Class** The event class to be interpreted.

**Filter** A string used for filtering events at event handling time. If the event name contains `Filter` as an initial substring, the event is handled, otherwise it is not handled.

**Location** Location of the event interpreter. Can be either E_LOCAL or E_REMOTE. If the location is E_LOCAL, the `Init()`, `Every()` and `Final()` functions must be defined in the client's name space. If the location is E_REMOTE, the client tries to connect to an event interpreter with the service name Service who provides three functions `Init()`, `Every()` and `Final()`[9]. The event interpreter does not have to be started up when

[9] In the remote case the names do not have to be identical.

`Event_Lookup_Interpreter` is executed.

RETURNS: If successful, it returns a client process unique id of the event interpreter, to be used as parameter in other event interpreter functions. If unsuccessful, it returns NO_EI_ID. A lookup is unsuccessful, if the event interpreter cannot be found or if the event class is unknown. If the call fails and event verbosity is on, an error message is printed.

NOTES: `Event_Lookup_Interpreter()` connects the client program to a local or remote event interpreter. If the event interpreter is local, the `Init()`, `Every()` and `Final()` functions must be defined in the client program. If the event interpreter is remote, it is looked up under Service in the name server domain BEE. If the remote event interpreter cannot be found after several attempts, the client proceeds without connection. The number of connection attempts can be controlled with `Event_Lookup_Retry()`.

**Event_Lookup_Remote_Interpreter**

INTERFACE:

```
int Event_Lookup_Remote_Interpreter(Service,
                                    Class,
                                    Filter)
char * Service;
Event_class_t Class;
char * Filter;
```

PARAMETERS:

| | |
|---|---|
| **Service** | Service name of event interpreter. |
| **Class** | The event class to be interpreted. |
| **Filter** | A string used for filtering events at event handling time. If the event name contains `Filter` as an initial substring, the event is handled, otherwise it is not handled. |

RETURNS: A client process unique id of the event interpreter, to be passed as parameter to other event interpreter functions.

NOTES: `Event_Lookup_Remote_Interpreter()` is a macro for

```
Event_Lookup_Interpreter(Name, 0,0,0, Class, Filter, E_REMOTE);
```

**Event_Disable_Interpreter**

Disable an event interpreter.

INTERFACE:

```
Event_return_t Event_Disable_Interpreter (EI_ID)
int EI_ID;
```

PARAMETERS:

| | |
|---|---|
| **EI_ID** | Event Interpreter Descriptor. |

RETURNS: E_SUCCESS if successful, E_FAILURE otherwise.

NOTES: Stop sending events to event interpreter until it is enabled again.

**Event_Enable_Interpreter**

Enable an event interpreter.

INTERFACE:

```
Event_return_t Event_Enable_Interpreter (EI_ID)
int EI_ID;
```

PARAMETERS:

| | |
|---|---|
| **EI_ID** | Event Interpreter Descriptor. |

RETURNS: E_SUCCESS if successful, E_FAILURE otherwise.

NOTES: Resume sending events to event interpreter if it is attached to the client.

**Event_Detach_Interpreter**

Remove event interpreter from client.

INTERFACE:

```
Event_return_t Event_Detach_Interpreter (EI_ID)
int EI_ID;
```

PARAMETERS:

**EI_ID** Event Interpreter Descriptor.

RETURNS: E_SUCCESS if successful, E_FAILURE otherwise.

NOTES: Execute the `Final()` function of an event interpreter for all event classes it has been attached to and delete it from the current event table.

## 8.8 Event Service Functions

Event name service functions make BEE services known to the name server.

**Event_EI_Server**

Start the EI server.

INTERFACE:

```
void Event_EI_Server(EI)
Event_interpreter_t EI;
```

PARAMETERS:

**EI** A record containing the path name of the executable and the `Init()`, `Every()` and `Final()` functions of the service.

RETURNS: `Event_EI_Server()` does not return.

DIAGNOSTICS:

**`"Event interpreter ready."`**

The EI server is running and waiting for commands from other clients or event interpreters.

`"Event interpreter not ready."`

The EI server could not be started up. (Often this means that the name server is not running).

NOTES: `Event_EI_Server()` assumes that the name server is running. `Event_EI_Server()` is called by `Event_Enter_Interpreter()`.

**Event_Client_Server**

Start the client server.

INTERFACE:

```
void Event_Client_Server()
```

PARAMETERS: None.

RETURNS: Nothing.

DIAGNOSTICS: In verbose mode, one of the following messsages is printed:

**`"Client server ready."`**

The client server is running and waiting for requests from other clients or event interpreters.

**`"Client server not ready."`**

The client server cannot be started up. This usually means that the global name server is not

running.

NOTES: The name of the event client server is the concatenation of the string "Client_Server_", the name of the client executable and the hexadecimal numbers of the node id and process id. On UNIX, the node id is the internet address of the host, on NECTAR it is the NECTAR node id. On NECTAR, the client server is a thread forked off by `N_Create_Appl()`. `Event_Client_Server()` assumes that the name server is running.

**Event_Enter_Service**

Make a service known to the name server and return its BEE port.

INTERFACE:

```
BEE_port_t Event_Enter_Service(S, D, Mode)
char * S;
char * D;
Event_lookup_mode_t Mode;
```

PARAMETERS:

| | |
|---|---|
| **S** | Name of the service to be entered into the name server. |
| **D** | Name server domain. Must be either "BEE" or "CLIENT". |
| **Mode** | What to do if service already exists. |
| | E_REPLACE Replace existing service in Bee domain. |
| | E_ERROR_IF_EXISTS Return illegal port NO_BEE_PORT if service exists. |
| | E_ERROR_IF_NOT_EXISTS Return illegal port NO_BEE_PORT if service does not exist. |

RETURNS: A local port to the service.

NOTES: Event interpreter services must be registered in the name server domain "BEE", client servers must be registered in the domain "CLIENT".

**Event_Lookup_Service**

Look for service S in name server domain D.

INTERFACE:

```
BEE_gl_port_t Event_Lookup_Service(S, D)
char * S;
char * D;
```

PARAMETERS:

| | |
|---|---|
| **S** | Name of the service to be looked up. |
| **D** | Name server domain, which must be either "BEE" or "CLIENT". |

RETURNS: A global port to the service. The port is either a client server port (domain "CLIENT") or an EI server port (domain "BEE").

NOTES: The name of the service is only known if it was registered previously with `Event_Enter_Service()`.

**Event_Attach_Client_Port**

Attach event interpreter to a client specified by a port.

INTERFACE:

```
Event_Attach_Client_Port(Port)
BEE_port Port;
```

PARAMETERS:

| | |
|---|---|
| **Port** | The event server port of the client to be controlled. |

RETURNS: E_SUCCESS if successful, E_FAILURE otherwise.

## 8.9 Event Interpreter Control Functions

These functions are issued by the client to provide hints or to control the behavior of attached event interpreters.

**Event_Replay**

Read events from a file.

INTERFACE:

```
void Event_Replay(File)
char *  File;
```

PARAMETERS:

| | |
|---|---|
| **File** | Name of an event file. The event file is generally produced by the event filer, but it can also be prepared manually by the programmer as long as it conforms to the event file format described in Section 9.10, page 59. |

RETURNS: Nothing.

NOTES: `Event_Replay()` replays a set of events produced in an earlier execution and therefore must be called before `Event_Enter_Enterpreter()`. This is automatically done if the environment variable BEE_REPLAY is set to the name of the event file.

**Event_Update_Rate**

Set the event interpreter update rate.

INTERFACE:

```
void Event_Update_Rate(Nr)
int Nr;
```

PARAMETERS:

| | |
|---|---|
| **Nr** | Update the event interpreter window associated with the client every at least every Nr events. Default value: 1. |

RETURNS: Nothing.

NOTES: `Event_Update_Rate()` can be called any time. However, when an event interpreter is looked up, the value of the update time for this event interpreter is determined by the latest `Event_Update_Rate()` call.

**Event_Update_Time**

Set the time between event interpreter window updates.

INTERFACE:

```
void Event_Update_Time(Time)
int Time;
```

PARAMETERS:

| | |
|---|---|
| **Time** | Update the screen every Time seconds. Default value: 3. |

RETURNS: Nothing.

NOTES: `Event_Update_Time()` can be called any time. When an event interpreter is looked up, the value of the update time for this event interpreter is determined by the latest `Event_Update_Time()` call.

**Event_Sample_Time**

Time interval for sampling event probes in the client.

INTERFACE:

```
void Event_Sample_Time(Time)
int Time;
```

PARAMETERS:

| | |
|---|---|
| **Time** | Sample the client event probes every **Time** seconds. Default: No sampling. |

RETURNS: Nothing.

NOTES: `Event_Sample_Time()` can be called any time. When an event interpreter is looked up, the value of the sample time for this event interpreter is determined by the latest `Event_Sample_Time()` call.

**Event_Display**

Name of the workstation to be used for display.

INTERFACE:

```
void Event_Display(Name)

char * Name;
```

PARAMETERS:

| | |
|---|---|
| **Name** | Name of display to be used by event interpreter when opening windows. For the X window manager, N has to be in the format **host:server:screen.** |

NOTES: `Event_Display()` can be called any time. **Name** is passed as a hint to the `Init` function of the event interpreter. `Event_Display()` therefore has to be called by the client before the event interpreter is looked up. The event interpreter will use the hint if it has not yet opened a window, otherwise the hint will be ignored.

**Event_Window**

Window configuration hints for event interpreter.

INTERFACE:

```
void Event_Window(Window_name,
                  Upper_left_x,
                  Upper_left_y,
                  Width,
                  Height,
                  Text_color,
                  Line_color)
char * Window_name;
int  Upper_left_x;
int  Upper_left_y;
int  Width;
int  Height;
int  Text_color
int  Line_color;
```

PARAMETERS:

| | |
|---|---|
| **Window_name** | Name of window and icon used by event interpreter. Default value: "Event EI". |
| **Upper_left_x** | Upper left x coordinate of window body. Default value: 1. |
| **Upper_left_y** | Upper left y coordinate of window body. Default value: 1. |
| **Width** | Width of window. Default value: 400. |
| **Height** | Height of window. Default value: 300. |
| **Text_color** | Color of text in the window. **Text_color** is not used if the screen is black-and-white. Default value: BLUE. |
| **Line_color** | Color of lines in the window. **Line_color** is not used if the screen is black-and-white. Default value: STEELBLUE. |

RETURNS: Nothing.

NOTES: If BEE is running under X, the upper left position specifies the position of the body of the window, not the position of the titlebar. The window information is made available to the event interpreter when the `Init()` function is called. `Event_Window()` therefore has to be called by the client before the event interpreter is looked up. BEE's available colors are defined in **$BEE_DIR/bee_color.h**.

**Event_Font**

Name of type font for text.

INTERFACE:

```
void Event_Font(F)
char * F;
```

PARAMETERS:

**F** Name of the type font used by event interpreter when displaying text in the X window.

RETURNS: Nothing.

NOTES: The font information is made available to the event interpreter when the `Init()` function is called. `Event_Font()` therefore has to be called by the client before the event interpreter is looked up. BEE has its own ideas about typefonts and linewidth when the window gets very small. It selects a 5x8 type font and also restricts the smallest window to a size depending on the view: The smallest linegraph view is a rectangle of 300x100 pixels, the smallest piechart view is a quadrant of 100x100 pixels.

**Event_Histogram_View**

Set the layout information for the histogram view.

INTERFACE:

```
void Event_Histogram_View(Title,
                          X_axis_name,
                          Y_axis_name,
                          X_axis_type,
                          Y_axis_type,
                          X_axis_limit,
                          Y_axis_limit)
char  * Title;
char  * X_axis_name;
char  * Y_axis_name;
axis_type_t X_axis_type;
axis_type_t Y_axis_type;
long  X_axis_limit;
long  Y_axis_limit;
```

PARAMETERS:

**Title** Title of view. Default value: "Event Profile".

**X_axis_name** String attached to lower right corner of the view. Default value: "".

**Y_axis_name** String attached to upper left corner of the view. Default value: "".

**X_axis_type, Y_axis_type**

For histogram views the axis type can be one of the following values:

| | |
|---|---|
| **LINEAR_NO_NUMBERS** | **Linear axis with no markers** |
| **LINEAR** | **Linear axis with numbered markers.** |
| **LINEAR2** | **Linear axis with unnumbered markers.** |
| **LOGARITHMIC** | **Logarithmic axis with numbers.** |

**X_axis_limit** Maximum sensor value allowed for display. Sensor values larger than **X_axis_limit** are represented by smaller columns.

**Y_axis_limit** Maximum number of event sensors. If the client enters more than

**Y_axis_limit**, only the **Y_axis_limit** most recently updated sensors are shown. This parameter is ignored in version BEE 2.7 and earlier versions.

RETURNS: Nothing.

NOTES: The layout information is made available to the event interpreter when the `Init()` function is called. `Event_Histogram_View()` therefore has to be called by the client before the event interpreter is looked up. BEE changes to a small typefont if the histogram window is small or if the number of sensors becomes so large that their name positions overlap each other in the current type font.

**Event_Linegraph_View**

Set the layout information for the linegraph view.

INTERFACE:

```
void Event_Linegraph_View(Title,
                          X_axis_name,
                          Y_axis_name,
                          X_axis_type,
                          Y_axis_type,
                          X_limit,
                          Y_limit,
                          Sliding_window,
                          Stack_curves)
char  *      Title;
char  *      X_axis_name;
char  *      Y_axis_name;
axis_type_t  X_axis_type;
axis_type_t  Y_axis_type;
long         X_limit;
long         Y_limit;
int          Sliding_window;
boolean_t    Stack_curves;
```

PARAMETERS:

**Title** Title of view. Default value: "Event Profile".

**X_axis_name** String attached to lower right corner of the view. Default value: "".

**Y_axis_name** String attached to upper left corner of the view. Default value: "".

**X_axis_type, Y_axis_type**

The y axis type should be one of the following:

| | |
|---|---|
| LINEAR | Linear axis with numbered markers showing absolute sensor values. |
| PERCENT | Axis with number markered showing percentages from 0 to 100. Sensor values at each time point always add up to 100%. |

The x axis currently always displays time in seconds. Thus the x axis type should be LINEAR or LINEAR2.

**X_limit** Maximum timestamp value (in secs). When an event with a time stamp larger than **X_limit** is encountered, the x axis is not rescaled.

**Y_limit** Maximum event sensor value. When an event sensor value larger than **Y_limit** is encountered, the y axis is not rescaled. (Y_limit is ignored if `Y_axis_type` is set to PERCENT).

**Sliding_window** Time range (in seconds) shown at the X axis.

**Stack_curves** If TRUE, each sensor line serves as the x axis of the next sensor line, with the exception of the first sensor which uses the x axis of the coordinate system as base line. If FALSE, all event sensor values are drawn with

respect to the x axis of the coordinate system.

RETURNS: Nothing.

NOTES:

- If the number of event sensors is large, rounding errors occur in the linegraph window: The sum of all event sensor values will be displayed as greater than 100%.
- The layout information is made available to the event interpreter when the `Init()` function is called. `Event_Linegraph_View()` therefore has to be called by the client before the event interpreter is looked up. The initial time at the origin is determined by the time stamp of the first incoming event. If it is between 0 and `Time_Window`, the time at origin starts with 0, otherwise it is chosen such that the event sensor value can immediately be displayed without rescaling the x axis.
- A client timestamp is always converted into event interpreter time by `Event_Virtual_Client_Time()` (See Section 8.10, 47) . When the event time advances past the time at the end of the x axis, the x axis is shifted. The size of the shift depends on the new event time. If it is less than the last encountered event time plus `Sliding_window`, the x axis is shifted 3/4 to the left, otherwise the time at the origin is computed in the same way as when the first time stamp was encountered. Because the incoming event streams are only partially ordered, there is also the possibility of *old* events, that is, events whose timestamp is older than the time at the origin. Old events are not displayed by the linegraph view ( the user is notified of old events if event verbosity is on).
- When event sensor values are encountered that exceed the current top value at the y axis, the y axis is rescaled. If possible, the new value is chosen such that the y scale markers are a multiple of 10.
- Event sensor values can only be positive.

## 8.10 Miscellaneous Functions

### Event_Virtual_Client_Time

Convert the time stamp of client event to event interpreter time.

INTERFACE:

```
void Event_Virtual_Client_Time(E);
Event_t E;
```

PARAMETERS:

E Client event.

RETURNS: Nothing.

NOTES: When a client from a remote node initially attaches to an event interpreter, BEE computes the difference between the time in the client's timestamp and the time of the event interpreter's clock (See `Event_Get_Time_Delta()`) and `Event_Virtual_Client_Time()` corrects E's time stamp by this difference. Of course, this method is only an approximation and it assumes a constant communication delay as well as the absence of clock drift. If the application is running for a very long time, clock drift becomes an issue and the offset between the clocks on different nodes will have to be recalculated periodically with `Event_Get_Time_Delta`.

### Event_Get_Time_Delta

Compute the difference between the timestamp in event E and the current clock.

INTERFACE:

```
boolean_t Event_Get_Time_Delta(E, Delta);
Event_t E;
int * Delta;
```

PARAMETERS: None.

RETURNS: TRUE if the time stamp is older than the clock, otherwise FALSE.

NOTES: The computed **Delta** is subsequently used by `Event_Virtual_Client_Time()`. `Event_Get_Time_Delta()` is automatically called by the EI server when a client initially attaches to an event interpreter.

**Event_Table_Inheritance**

Set the inheritance mode for event tables.

INTERFACE:

```
void Event_Table_Inheritance(Flag);
boolean_t Flag;
```

PARAMETERS:

**Flag** If TRUE, create a task event table whenever a task is created. The initial table is an exact copy of the system event table, but only the event classes are inherited, not the attached event interpreter lists. Events generated in a task are handled by the task event table, and not by by the system event table. If **Flag** = FALSE, the system event table is used for all events. The default value is **Flag** = FALSE.

RETURNS: Nothing.

NOTES: In NECTAR, `Event_Table_Inheritance(TRUE)` cannot be called in a Nectarine task. If `Event_Table_Inheritance(FALSE)` is called in a task, all events generated in that task are from then on handled by the system event table.

**Event_Remote_Print**

Redirect the output of `PRINT_MSG` macros.

INTERFACE:

```
void Event_Remote_Print (Flag)
boolean_t Flag;
```

PARAMETERS:

**Flag** If TRUE, `PRINT_MSG()` macros send the string to be printed to the event interpreter service `"bee_print_msg"`. If FALSE, `PRINT_MSG()` prints the string into the active typescript window.

RETURNS: Nothing.

NOTES: Remote printing is only possible if print messages are not filtered and if the call to `PRINT_MSG()` is issued outside the event kernel. Remote printing generates event points of type E_STRING.

**Event_Init_Debug_Switches**

Debugging switches used for debugging of the event kernel.

INTERFACE:

```
void Event_Init_Debug_Switches()
```

PARAMETERS: None.

RETURNS: Nothing.

NOTES: The following debug switches are known:

| | |
|---|---|
| EH_DEBUG | Debug event handler functions. |
| E_SERVER_DEBUG | Debug message traffic between client and event interpreters. |

**Event_Version**

Print BEE's version number.

INTERFACE:

```
void Event_Version ()
```

PARAMETERS: None.

RETURNS: Nothing.

**Event_Protocol_Version**

Return version number of event protocol.

INTERFACE:

```
int Event_Protocol_Version ()
```

PARAMETERS: None.

RETURNS: Protocol number of event protocol.

NOTES: The event protocol version is changed whenever the event protocol implementation is changed.

**Event_Verbose**

Control verbosity when executing event kernel functions.

INTERFACE:

```
void Event_Verbose (Flag)
boolean_t Flag;
```

PARAMETERS:

| | |
|---|---|
| **Flag** | If TRUE, various event kernel functions are executed verbose. If FALSE, event kernel functions are executed silently. |

RETURNS: Nothing.

**Event_Lookup_Retry**

Number of retries when looking up an event interpreter.

INTERFACE:

```
void Event_Lookup_Retry (Retry)
int Retry;
```

PARAMETERS:

| | |
|---|---|
| **Retry** | Retry determines the number of times a client program to retry after an unsuccessful lookup of a remote event interpreter. -1 means forever. The default value is 1. |

RETURNS: Nothing.

NOTES: See `Event_Lookup_Remote_Interpreter()`.

**Event_Buffer_Size**

Size of event buffers allocated by the event kernel.

INTERFACE:

```
void Event_Buffer_Size (Nr_of_events)
int Nr_of_events;
```

PARAMETERS:

| | |
|---|---|
| **Nr_of_events** | Maximum number of events that can be be stored in the event buffer. The default value is EVENT_BUFFER_SIZE (currently set to 100). |

RETURNS: Nothing.

NOTES: In verbose mode (see `Event_Verbose()`), the new event buffer size and the size of one event are printed on standard output.

# 9 Event Protocol

BEE can be seen as a remote procedure call mechanism with filters optimized for the purpose of event processing. In this section we deal with the lowest level of BEE, its event protocol. Sections 9.1 to 9.6 describe the internal representation of event records and event interpreters. Section 9.7 explains the types of messages that can be exchanged between clients and event interpreters. Section 9.8 describes the event network format used for exchanging events between heterogenous machines and for archiving events and Section 9.9 lists the access functions defined on event messages.

## 9.1 Event Class

Bee offers a small set of predefined event classes. In addition, it provides the user with the ability to define user defined classes. The predefined event classes are:

```
typedef enum event_class {
                          E_PROCEDURE,
                          E_EVENT,
                          } Event_predefined_class_t;
```

E_PROCEDURE — This event class is generated with the BEGIN/END sensors defined in section 8.2. Because this event class does not contain an event variant, the name of the event sensor is piggybacked in the event variant. To reduce the client overhead and network traffic, this is done only the first time the sensor encountered.

E_EVENT — A general event range class with no variant part. Generated by the BEGIN_EVENT/END_EVENT sensors defined in section 8.2.

Another event class, E_STRING is predefined by BEE, but only entered into the event table if the client is attached to the remote print service:

E_STRING — An event with a string in the variant part. Generated if remote printing is on (see section 8.10).

The constant E_CLASSMAX denotes the maximum number of predefined and userdefined event classes.

## 9.2 Event Attributes

```
typedef enum event_attribute
        {E_ACT,
         E_TERM,
         E_POINT,
         E_AGGREGATE,
         E_NAME,
         E_PROBE} Event_attribute_t;
```

E_ACT — The activation of an event range.

E_TERM — The termination of an event range.

E_POINT — An event point.

E_AGGREGATE — An aggregate event.

E_NAME — The name of an event sensor (contained in the variant part).

E_PROBE — An event probe.

## 9.3 Event Record

The predefined data type **Event_record_t** is declared as follows:

```
#define E_DATA_LEN <implementation dependent>/* Maximum event variant size */

typedef struct Time_val {
        long seconds;
        long ticks;   /*
                       * On a Unix host node, a tick is equal
                       * to a microsecond.  On the Nectar CAB, a tick
                       * is equal to 960 nanoseconds.
                       */
} * Time_val_t;

typedef int Event_class_t;  /* Unique identifier for event class */

typedef int Event_id_t;     /* Unique identifier for event sensor */


typedef struct Event_record {
   struct Time_val TS;          /* Timestamp in seconds, milliseconds       */
   Event_id_t EID;              /* Event sensor descriptor (Process unique) */
   Event_class_t Class;         /* Event class                              */
   Eventa_ttribute_t Attr;      /* Event attribute                          */
   Event_node_t NodeID;         /* Node descriptor (Network unique)         */
   Event_node_t ProcessID;      /* Process descriptor (Node unique)         */
   Event_thread_t ThreadID;     /* Thread Id in process (Process unique)    */
   union {                      /* Event class specific variant             */
     char   String[E_DATA_LEN];/* String in E_STRING event                 */
     char   Data[E_DATA_LEN];  /* For user defined events                  */
      } Variant;
   } * Event_record_t;
```

The fields of an event record are:

**TS** The time stamp describes the time of the event when it was generated on the client side. The time is expressed in elapsed seconds and microseconds since 00:00 GMT, January 1, 1970 (zero hour). The resolution of the system clock is hardware dependent; the time may be updated continuously or in ticks. On NECTAR, the time is read from a register on the CAB and provides a resolution of 1 µs. The Cray clock has nanosecond resolution, but for portability reasons, BEE rounds Cray time stamps to microseconds. Under UNIX, the time stamp is never correct enough that one should believe the microsecond values. On Sun-4 systems the clock resolution is 1 msec. The clock of a VAX 3100 has a resolution of 10 msec and on Sun-3 systems it is 20 msec.

**EID** The event sensor descriptor or event id. Event id's are small integers assigned by the event kernel and are unique on a process basis.

**Class** A small integer describing the class of the event (See section 9.1).

**Attribute** A small integer describing the attribute of the event. Possible values are described in section 9.2.

**NodeID** The node id of a host on which the process is running. It is unique for each node in the network. BEE' TCP/IP implementation uses the internet address of the host as node id.

**ProcessID** The process id of a client process. It is unique for each workstation, however different workstations can assign the same process id to different client programs generating events.

**ThreadID** A small integer used to distinguish multiple threads in one client process.

**Variant** The data dependent part of an event depends on the **Class** as well as on the **Attribute** field. For example, if the attribute is E_NAME, the variant contains the name of the event sensor. When an event sensor of type E_PROCEDURE is encountered the first time, the variant also contains the name of the sensor, that is, the name is "piggybacked" on the first event with the attribute E_ACT. In the case of an user defined event, the type of the variant is described by an event message of type E_DESCRIPTOR (See Section 9.8.1).

## 9.4 Event Table

The event table is an array of E_CLASSMAX event associations, where the value of E_CLASSMAX is implementation dependent. An *event association* describes an event class and the event interpreters attached to it.

```
typedef struct event_association {
  Event_class_t Class;
  char Name[E_NAME_LEN];
  boolean_t Enabled;              /* TRUE: generate events for class */
  Event_interpreter_list_t EI; /* List of event interpreters      */
}  * Event_association_t ;
```

BEE supports two kinds of event tables: system and task event tables. The system event table is generated at client startup time and contains the event associations for the predefined event classes. Task event tables are associated with threads or Nectarine tasks and are generated at thread/task creation time. They inherit the event classes defined in the system event table at that time. Whenever an event is generated the event handler selects the appropriated event table and scans the event associations for associated event interpreters. The creation of task event tables can be suppressed with `Event_Table_Inheritance()`. In this case all event sensors are processed in the context of the system event table.

## 9.5 Client Server and EI Server Port

BEE knows about two types of event servers. The client server is created for each client program and is accessible via the client server port `event_client_port`. The client server accepts event kernel requests from remote event interpreters via the EI server port `event_server_port`. The EI server is a module or thread started by `Event_Enter_Interpreter()`. The EI server accepts events from attached clients. Attachment is done with `Event_Lookup_Interpreter()` by requesting the EI server port from the name server given the name of the event interpreter service.

```
#define BEE_port_t    <implementation dependent>  /* Node unique port    */
#define BEE_gl_port_t <implementation dependent>  /* Network unique port */

BEE_gl_port_t event_client_port; /* Client server port
                                    of current client*/
BEE_gl_port_t event_server_port; /* EI server port */
```

## 9.6 Event Interpreter

An event interpreter is represented internally as:

```
/*
 *  Event Interpreter Lookup mode
 */
typedef enum event_lookup_mode {
      E_REPLACE,              /* Replace existing service in Bee domain */
      E_ERROR_IF_EXISTS,      /* Raise error if service exists */
      E_ERROR_IF_NOT_EXISTS /* Raise error if service does not exist */
     } Event_lookup_mode_t;
/*
 *  Client death mode
 */
typedef enum event_client_death_mode {
      E_DELETE_SENSORS,     /* Delete all information about client  in
                               event sensor array */
      E_KEEP_SENSORS       /* Keep event sensor information about client */
     } Event_client_death_mode_t;
```

```
typedef struct Event_interpreter {
     char Name[MAXPATHLEN];
     Event_return_t (* Init) ();
     Event_return_t (* Every) ();
     Event_return_t (* Final) ();
     char Filter[E_NAME_LEN];
     Event_lookup_mode_t EI_lookup_mode;
     Event_client_death_mode_t Client_death_mode;
 } *  Event_interpreter_t;
```

**Name** is the path name of the executable client program. **Init, Every, Final** are pointers to functions with the following interface:

```
Event_return_t Init(E) Event_t E;

Event_return_t Every(E) Event_t E;

Event_return_t Final(E) Event_t E;
```

The string **Filter** is used by the event handler to send only those events to attached event interpreters whose event sensor names have **Filter** as a prefix. The **EI_lookup_mode** specifies what to do if the service exists or does not exist in the name server domain. And **Client_Death_mode** specifies what to do with the accumulated event sensor information in the case of a client death.

## 9.7 Message Types

In BEE's event processing model, the communication between client and event interpreters is bi-directional. Messages sent from the client to the event interpreter are called *events* and the delivery of an event is guaranteed by the underlying communication system. In the NECTAR implementation of BEE, events are sent with NECTARINE'S reliable message protocol, in the UNIX implementation they are sent with sockets using TCP/IP. The connection between client and event interpreter for events is done with the `Event_Lookup_Interpreter()` and `Event_Enter_Interpreter()` functions via the name server. Events are described in section 9.8.1.

Communication between event interpreter and client is done by a request-response protocol which is always initiated by the event interpreter. Messages sent from the event interpreter to the client are called *event interpreter requests* (or simply requests). The event interpreter issues the request to the client and indicates a reply port. The client then executes the corresponding event kernel function and sends a reply message containing the return result to the indicated reply port. The event kernel function `Event_Attach_Client_Port()` attaches an event interpreter to a particular client. If the event configuration is planned, the client server port is made known to the event interpreter with the INIT event. In an unplanned event configuration, the client server port can be determined with `Event_Lookup_Service()`. Event interpreter requests and client replies are described in section 9.8.2 and section 9.8.3, respectively.

## 9.8 Event Network Format

The event network format specifies the format of messages exchanged between clients and event interpreters. `Network_Format_t` is an enumerated type, but for portability reasons, it is of type string. Currently the following values are known:

```
#define E_RESERVED_FORMAT   "0"  /* Indicates illegal network format   */
#define E_ASCII_FORMAT      "1"  /* Message is in ASCII format.        */
#define E_SUN_FORMAT        "2"  /* Message in native Sun format       */
#define E_VAX_FORMAT        "3"  /* Message in native Vax format       */
#define E_CRAY_FORMAT       "4"  /* Message in native Cray format      */

typedef char Event_network_format_t[4];
```

### 9.8.1 Events

The general format of an event is

```
typedef enum E_client_cmd {
          E_INIT,           /* Initialization message          */
          E_EVERY,          /* Every message                   */
          E_FINAL,          /* Final message                   */
          E_REGISTRATION,   /* Client registration             */
          E_DEATH,          /* Client death                    */
          E_REQUEST,        /* Event interpreter request       */
          E_REPLY,          /* Client reply                    */
          E_DESCRIPTOR      /* Type descriptor of event variant */
          } E_client_cmd_t;

typedef struct Event_msg {
          Event_network_format_t  Format;
          E_client_cmd_t Cmd;
          struct Event_record Event;
            union {
              Event_init_t Init;
              Event_client_t EventServer;
            } data;
} * Event_t ;
```

The contents of the data field depends on the value of Cmd of the event message:

E_INIT — Sent to an event interpreter immediately after the client registration message has been sent. Contains client's recommendation about desired layout and update frequency of event information. The Event_init_t structure is defined as follows:

```
typedef struct bee_display_info {
  char Title_name[E_TITLE_NAME]; /* graph  title            */
  char Xaxis_name[E_XAXIS_NAME]; /* name of x-axis          */
  char Yaxis_name[E_YAXIS_NAME]; /* name of y-axis          */
  char Xaxis_unit[E_XAXIS_UNIT]; /* unit of x-axis,         */
  axis_type_t Xaxis_type;        /* type of xaxis           */
  long Xaxis_limit;              /* Maximum x value         */
  long Yaxis_limit;              /* Maximum y value         */
  long X_init;                   /* Initial x-axis value.   */
  long Y_init;                   /* Initial y-axis value.   */
  int  Sample_Time;              /* Time between  samples   */
  int  Update_Rate;              /* display update rate     */
  int  Update_Time;              /* display update time     */
  long Sliding_window;           /* width of x axis in secs */
  boolean_t Stacking_mode;       /* TRUE:use baseline of
                                    previous sensor*/
  int  Text_color;               /* Color used for text     */
  int  Line_color                /* Color used for  lines   */
  } bee_display_info_t;

typedef struct bee_window_info {
  char Display[E_DISPLAY_NAME];  /* X display               */
  int  Upper_left_x;
  int  Upper_left_y;
  int  Window_width;
  int  Window_height;
  char Window_name[E_WIN_NAME];  /* name of X-window        */
  char Font_name[E_FONT_NAME];   /* Font to be used by EI   */
  } bee_window_info_t;

typedef struct event_init {
        bee_window_info_t Window;
        bee_display_info_t Display;
} Event_init_t ;
```

E_FINAL — Sent to any active event interpreter when client calls Event_Death_Interpreter(). The

E_FINAL message is of type `Event_msg` (defined in Section 9.8.1, page 55) with an empty `data` field.

E_REGISTRATION This event type is sent to the event interpreter when client creates a connection as the result of a `Event_Lookup_Interpreter()` call. In the `Event_record` field only `Class` is initialized. The variant part of the E_REGISTRATION message is defined as follows:

```
typedef struct event_client {
   boolean_t Active;                /* client has a valid event
                                       client server port       */
   char Client_name[E_MAXPATHLEN];/* Name of client's runfile*/
   Event_id_t Node;                 /* Node id of client        */
   long Process;                    /* Process id of client     */
   long Thread;                     /* Thread of client         */
   Event_machine_t Machine;         /* Processor type on which
                                       client executes.         */
   BEE_gl_port_t Port;              /* Event client server port*/
   BEE_port_t Reply_port;           /* Reply port for requests */
   long Nr_of_conns;                /* Open client connections */
   struct Time_val Time_delta;      /* Difference between EI time
                                       and client time          */
   boolean_t Client_time_ahead;  /* TRUE: (Client time-EI time)
                                       >= 0, FALSE otherwise.   */
} Event_client_t;
```

Only the fields `Name`, `Node` and `Port` are filled by the client, the other fields are filled by the EI server. `Time_Delta` is the difference between the local event interpreter time and the time stamp of the registration messsage. It is added to all event time stamps generated by the client to convert client time to EI time. BEE assumes that clocks on different hosts do not have the same time, but do not drift in relation to each other during the client run.

E_DEATH Sent to attached event interpreter when the client calls `Event_Delete_Interpreter()` or sent to any attached event interpreter when the client dies or finishes execution (only, of course, if the client's death can be diagnosed). The E_CLIENT_DEATH message is of type `Event_msg` (defined in Section 9.8.1, page 55) with an empty `data` field.

E_DESCRIPTOR For each user defined class, a message of type E_DESCRIPTOR is sent to the attached event interpreter when `Event_Lookup_Interpreter()` is called. The `Class` field in the `Event_record` describes the event class. The variant part of the E_DESCRIPTOR message contains the type descriptor describing the layout of the user defined event:

```
typedef struct event_variant_descriptor {
        Typedescriptor[E_MAXPATHLEN];
} Event_variant_descriptor_t ;
```

The type descriptor uses the control string syntax from C's I/O facility. For example, the type descriptor `"%d%d%f%d%s"` describes an event variant consisting of two integers, a floating point number, another integer and a string.

### 9.8.2 Event Interpreter Requests

The format of an EI request is:

```
typedef struct event_EI_request {
        Event_network_format_t  Format;
        E_EI_cmd_t Cmd;
        long ID;
        } * Event_EI_request_t;
```

Only a subset of the event kernel functions are available as requests.The possible requests are:

```
typedef enum E_EI_cmd {
                        E_CLASS_NAME,
                        E_ENABLE_CLASS,
                        E_DISABLE_CLASS,
                        E_DETACH_INTERPRETER,
                        E_DISABLE_INTERPRETER,
                        E_DELETE_INTERPRETER,
                        E_ENABLE_INTERPRETER,
                        E_KILL,
                        E_SENSOR_NAME,
                        E_DESCRIPTOR,
                        E_VERBOSE,
                        E_VERSION
                      } E_EI_cmd_t;
```

E_CLASS_NAME Get print name of event class. The `ID` field contains the event class id.

E_ENABLE_CLASS Enable processing of events of the specified class which is contained in the `ID` field of the message.

E_DISABLE_CLASS Disable processing of events of a specified class which is contained in the `ID` field of the message..

E_DETACH_INTERPRETER
Detach event interpreter. The `ID` field of the message contains the event interpreter descriptor.

E_DISABLE_INTERPRETER
Disable event interpreter. The `ID` field contains the event interpreter descriptor.

E_DELETE_INTERPRETER
Delete event interpreter. The `ID` field contains the event interpreter descriptor.

E_ENABLE_INTERPRETER
Enable event interpreter. The `ID` field contains the event interpreter descriptor.

E_KILL Acknowledgement by the event interpreter that it no longer wants to communicate with the client server. This message is only sent by the event interpreter after it received a E_FINAL message. The client attached to event interpreters cannot exit before having received an E_KILL message.

E_SENSOR_NAME Get print name of event sensor. The `ID` field contains the event sensor id.

E_DESCRIPTOR Get type descriptor for user defined event class. The `ID` field contains the event class id.

E_VERBOSE Control event processing verbosity in client.

E_VERSION Get BEE and protocol version.

### 9.8.3 Client Replies

The format of a client *reply* is:

```
typedef struct event_client_reply {
        Event_network_format_t  Format;
        Event_return_t Return_code;
        char Info[E_MAXPATHLEN];
      } * Event_client_reply_t;
```

The `Return_code` field indicates success (E_SUCCESS) or failure (E_FAILURE) of the request. In the case of a lookup, the `Info` field contains the name of the event class or event sensor. If the EI request was E_VERSION, `Info` contains BEE's version number and the protocol version separated by a blank space.

## 9.9 Event Access Functions

Event access functions permit to access individual fields in an event. In the following we describe functions that access the general fields as well as variant fields that depend on the event class.

### 9.9.1 General Access Functions

First we describe event access functions that are defined for all event messages:

**(int) E_Type(E)** Type of event E. The type is any scalar defined by `E_client_cmd_t` or `E_EI_cmd_t`.

**(pointer_t) E_Timestamp(E)**
Pointer to time stamp of event E. In BEE 2.6, time stamps are not unique. The time is read from the CAB clock, a 32 bit integer with a micro second resolution which wraps around approximately every hour.

**(long) E_Seconds(E)**
Seconds in time stamp of event.

**(long) E_Ticks(E)** Ticks in time stamp of event. On a Unix host node, a tick is equal to a microsecond. On a Nectar CAB, a tick is equal to 960 nanoseconds.

**(int) E_Class(E)** Event class of event.

**(int) E_Process_Id(E)**
Process id of event. If the event occurred in a Unix process, it is the Unix process number of the client, if the event occurred on a Cab, it is the Nectar id of the CAB processor.

**(int) E_Thread_Id(E)**
Thread id in which the event occurred. In NECTAR, the Nectarine Task id in which the event occurred.

### 9.9.2 E_INIT Access Functions

For events of type E_INIT the following functions are defined:

**(char *) EI_Window_Name(E)**
Name of event interpreter(EI)'s X window.

**(int) EI_Upper_Left_X(E)**
Upper left x coordinate of EI's X window (not the titlebar!).

**(int) EI_Upper_Left_Y(E)**
Upper left y coordinate of EI's X window (not the title bar).

**(int) EI_Height(E)** Height of EI's X window.

**(int) EI_Width(E)** Width of EI's X window.

**(char *) EI_Title(E)**
Main title to be used by EI.

**(char *) EI_Xaxis(E)**
Title to be attached to X axis.

**(char *) EI_Yaxis(E)**
Title to be attached to Y axis.

**(int) EI_Xunit(E)** Number of sub units on X axis.

**(int) EI_Yunit(E)** Number of sub units on Y axis.

**EI_Xtype(E)** Scale of X axis: Linear or logarithmic.

**EI_Ytype(E)** Scale of Y axis: Linear, percent or logarithmic.

**(long) EI_Xlimit(E)**
Maximum x value.

**(long) EI_Ylimit(E)**
Maximum y value.

**(int) EI_Sample_Time(E)**
Time interval for sampling event probes.

**(int) EI_Update_Rate(E)**
Maximum number of events between view updates.

**(int) EI_Update_Time(E)**
Time interval between view updates.

**(int) EI_Time_Window(E)**
Range to be used when displaying time.

**(int) EI_Text_Color(E)**
Color of textual information such as sensor names, main title, etc. Colors known by BEE are defined in `bee_color.h`.

**(int) EI_Line_Color(E)**
Color to be used for histogram columns, piecharts sectors and other lines. Colors known by BEE are defined in `bee_color.h`.

### 9.9.3 E_EVERY And E_FINAL Access Functions

The following event access functions are defined on events of type E_EVERY and E_FINAL:

**(int) E_Qual(E)** Denotes whether the event an event point, or the activation or termination of an event range.

**(int) E_Id(E)** Event sensor descriptor (unique only at process level).

**(long) E_Long_Value(E)**
Value contained in variant of E.

**(char *) E_String_Value(E)**
Pointer to a string contained in variant of E.

**(pointer_t) E_Pointer_Value(E)**
Pointer to an untyped value contained in variant of E.

### 9.9.4 E_REGISTRATION And E_DEATH Access Functions

For events of type E_REGISTRATION and E_DEATH we have:

**(char *) E_Client_Name(E)**
Name of client.

**E_Client_Process(E)**
Process Id of Client.

**E_Client_Port(E)** BEE port of client server.

### 9.9.5 E_DESCRIPTOR Access Functions

**(char *) E_Descriptor(E)**
Type descriptor for event class.

## 9.10 Event Network Format

The event protocol offers several message formats: an Ascii format and a set of host formats. In the Ascii format all the components of an event message are represented as text. If a client and an event interpreter are compiled by the same compiler and execute on machines of the same architecture they can exchange messages in host format. To determine whether boths parties can indeed share the same format, the event protocol contains a negotiation phase when the client connects to the event interpreter. As a result of the negotiation the client knows which

message format it can use.

For portability reasons, the event protocol describes only the Ascii format, but none of the host formats. Host formats are determined de facto by the compiler's allocation strategy for structures. Currently BEE supports host formats for Sun, Vax and Cray. New host formats can be added relatively easy by defining new scalars for the enumerated type `Event_Network_Format_t` and by providing conversion routines between the new format and the Ascii format. The Ascii format is used by various parts of BEE, such as the event filer, BEE's event replay facility and by the event kernel when exchanging events, requests and replies between heterogenous hosts.

### 9.10.1 Events, Requests and Replies

```
#define EVENT_ASCII_MSG_SIZE 511
typedef struct Event_Ascii_msg  {
        Event_network_format_t  Format;
        char Event_Message[EVENT_ASCII_MSG_SIZE];
      } * Event_Ascii_t;
```

**E_INIT**

The format of `Event_Message` for an event message of type E_INIT is:

```
(0  (<WindowFields>) (<DisplayFields>) (<EventRecord>))
```

The `<EventRecord>` fields are described in Section 9.3. `<WindowFields>` contains entries for each of the fields in `bee_window_info_t`, and `<DisplayFields>` contains entries for each of the fields in `bee_display_info_t` described in Section 9.8.1. The corresponding type descriptor is

```
(%d (%s %d %d %d %d) \
    (%s %s %s %d %d %d %d %d %d %d %d %d %d %d) \
    (%d %d %d %x %d %d %d.%d))
```

**E_EVERY and E_FINAL**

The format of `Event_Message` for event messages of type E_EVERY and E_FINAL, respectively, is:

```
(1 (<EventRecord>))

(2 (<EventRecord>))
```

The fields of `<EventRecord>` are described in section 9.3. The corresponding type descriptor is

```
(%d (%d %d %d %x %d %d %d.%d %s))
```

**E_REGISTRATION and E_DEATH**

The format of `Event_Message` for event messages of type E_REGISTRATION and E_DEATH, respectively, is:

```
(3 (ClientName Node Port) (<EventRecord>))

(4 (ClientName Node Port) (<EventRecord>))
```

`ClientName` is the full path name of the executable runfile. `Node` is a unique identification of the workstation on which the client is running. `Port` is the client's client server port. The fields of `<EventRecord>` are described in section 9.3. The corresponding type descriptor is

```
(%d (%s %x %d %d %x %d) (%d %d %d %x %d %d %d.%d))
```

**E_DESCRIPTOR**

The format of event entries of type E_DESCRIPTOR is

```
(7 (Type) (<EventRecord>))
```

`Type` cannot contain any blanks. The event class in `EventRecord` contains the event class for `Type`. The corresponding type descriptor is

```
(%d (%s) (%d %d %d %x %d %d %d.%d))
```

### 9.10.2 Event File Format

An event file consists of a header and a list entries in Ascii format:

```
(Bee_Version Protocol_Version)
(Cmd Event)
...
(Cmd Event)
```

The header describes the Bee version and the event protocol version of the event interpreter that created the event file. Each of the entries (Cmd Event) describe an event generated by a client, a request initiated by a event interpreter or a client reply. Entries in an event file are only partially ordered by the timestamps of the generating clients. Event files can be automatically produced by attaching the event interpreter "bee_filer"[10], but they can also edited manually. An example of an event file is shown below:

```
(BEE2.8:25-Oct-1990 0)
(3  (test 5506 8002fadc 2205 0 0) (1 1 2 16559 8002fadc 2205 0 4.945444 ))
(0 (counter 0 0 100 200) (Profile secs Percent 0 0 0 0 0 5 1 20 1 0 0) \
        (1 0 0 14473 8002fadc 2205 0 5.0 ))
(1 (1 0 0 14473 8002fadc 2205 0 5.0 foo ))
(1 (1 0 1 14473 8002fadc 2205 0 6.0 ))
(1 (1 0 0 14473 8002fadc 2205 0 13.510071 ))
(1 (1 0 1 14473 8002fadc 2205 0 13.510447 ))
(1 (9 0 0 16559 8002fadc 2205 0 27.858328 bar ))
(1 (9 0 1 16559 8002fadc 2205 0 27.858715 ))
(1 (10 0 0 16559 8002fadc 2205 0 27.858328 hello ))
(1 (10 0 1 16559 8002fadc 2205 0 27.858715 ))
(1 (9 0 0 16559 8002fadc 2205 0 27.862600 ))
(1 (10 0 0 16559 8002fadc 2205 0 27.862005 ))
(1 (10 0 1 16559 8002fadc 2205 0 27.862389 ))
(1 (9 0 1 16559 8002fadc 2205 0 27.863000 ))
(7 (%f) (5 4 2 16559 8002fadc 2205 0 29.743550))
(7 (%f) (6 4 2 16559 8002fadc 2205 0 29.743552))
(1 (5 4 2 16559 8002fadc 2205 0 29.743554 SensorA ))
(1 (5 3 2 16559 8002fadc 2205 0 29.743735 30 ))
(1 (6 4 2 16559 8002fadc 2205 0 29.743920 SensorB ))
(1 (6 3 2 16559 8002fadc 2205 0 29.744077 40.45 ))
(1 (7 4 2 16559 8002fadc 2205 0 29.744262 SensorC ))
(1 (7 3 2 16559 8002fadc 2205 0 29.744420 40 ))
(1 (5 3 2 16559 8002fadc 2205 0 193.883695 25 ))
(1 (6 3 2 16559 8002fadc 2205 0 193.883943 35.34 ))
(1 (7 3 2 16559 8002fadc 2205 0 193.884256 45 ))
(2 (-1 1 2 16559 8002fadc 2205 0 194.884256 ))
(4  (test 5506 8002fadc 2205 0 0) (-1 1 2 16559 8002fadc 2205 0 194.884256 ))
```

## 9.11 Protocol Interface

The following functions describe the interface of the event protocol.

**Event_Send_Init**

Send message E of type E_INIT reliably to event interpreter at port Port.

INTERFACE:

```
void Event_Send_Init(Port, E)
Bee_gl_port Port;
Event_t E;
```

PARAMETERS:

| | |
|---|---|
| **P** | Port of remote event interpreter. |

[10]Note that the event filer currently stores only event messages, but not event requests or event responses. Thus BEE is not able to replay the complete event related behavior of a client from the event file of an event filer.

| | | |
|---|---|---|
| | **E** | Init Message |

RETURNS: Nothing.

NOTES: **Event_connect_error** is set to E_FAILURE if event interpreter is unreachable.

**Event_Send_Every**

Send message E of type E_EVERY reliably to event interpreter at port **Port**.

INTERFACE:

```
void Event_Send_Every(Port, E)
Bee_gl_port Port;
Event_t E;
```

PARAMETERS:

| | |
|---|---|
| **Port** | Port of remote event interpreter. |
| **E** | Every Message |

RETURNS: Nothing.

NOTES: **Event_connect_error** is set to E_FAILURE if event interpreter is unreachable.

**Event_Send_Final**

Send message E of type E_FINAL reliably to event interpreter at port **Port**.

INTERFACE:

```
void Event_Send_Final(Port, E)
Bee_gl_port Port;
Event_t E;
```

PARAMETERS:

| | |
|---|---|
| **Port** | Port of remote event interpreter. |
| **E** | Final Message |

RETURNS: Nothing.

NOTES: **Event_connect_error** is set to E_FAILURE if event interpreter is unreachable.

**Event_Send_Registration**

Send message E of type E_REGISTRATION reliably to event interpreter at port **Port**.

INTERFACE:

```
void Event_Send_Registration(Port, E)
Bee_gl_port Port;
Event_t E;
```

PARAMETERS:

| | |
|---|---|
| **Port** | Port of remote event interpreter. |
| **E** | Registration Message |

NOTES: **Event_connect_error** is set to E_FAILURE if event interpreter is unreachable.

**Event_Send_Client_Death**

Send message E of type E_DEATH reliably to event interpreter at port **Port**.

INTERFACE:

```
void Event_Send_Client_Death(Port, E)
Bee_gl_port Port;
Event_t E;
```

PARAMETERS:

| | |
|---|---|
| **Port** | Port of remote event interpreter. |

| | | |
|---|---|---|
| | **E** | Client Death Message |

RETURNS: Nothing.

NOTES: **Event_connect_error** is set to E_FAILURE if event interpreter is unreachable.

**Event_Send_Descriptor**

Send message E of type E_DESCRIPTOR reliably to event interpreter at port **Port**.

INTERFACE:

```
void Event_Send_Descriptor(Port, E)
Bee_gl_port Port;
Event_t E;
```

PARAMETERS:

| | |
|---|---|
| **Port** | Port of remote event interpreter. |
| **E** | Type Descriptor Message |

RETURNS: Nothing.

NOTES: **Event_connect_error** is set to E_FAILURE if event interpreter is unreachable.

**Event_Receive_Msg**

Receive an event message **E** on port **Port**, report size of message in **Size**.

INTERFACE:

```
void Event_Receive_Msg(Size, Port, E)
int Size;
Bee_gl_port Port;
Event_t E;
```

PARAMETERS:

| | |
|---|---|
| **Size** | Size of received event message. |
| **Port** | Port of remote event interpreter. |
| **E** | Event Message |

RETURNS: Nothing.

NOTES: **Event_connect_error** is set to E_FAILURE if event interpreter is unreachable.

**Event_Send_Request**

Send a **Request** to the client server at **Port**, waiting for an answer on **ReplyPort**.

INTERFACE:

```
void Event_Send_Request(Port, Request, ReplyPort)
Bee_gl_port Port;
Event_EI_request_t  Request;
Bee_gl_port ReplyPort;
```

PARAMETERS:

| | |
|---|---|
| **Port** | Client server port. |
| **Request** | Event interpreter request. |
| **ReplyPort** | Port on which event interpreter expects the reply from the client. |

RETURNS: Nothing.

NOTES: **Event_connect_error** is set to E_FAILURE if client is unreachable.

**Event_Receive_Request**

Receive a **Request** at port **Port**, size of received message is placed in **Size** using the **ReplyID** specified when sending the request.

INTERFACE:

```
void Event_Receive_Request(Size, Port, Request, ReplyID)
int Size;
Bee_gl_port Port;
Event_EI_request_t  Request;
Bee_gl_port ReplyID;
```

PARAMETERS:

| | |
|---|---|
| **Size** | Size of received message. |
| **Port** | Event interpreter port. |
| **Request** | Event interpreter request. |
| **ReplyPort** | Port or ID on which event interpreter expects the reply from the client. |

RETURNS: Nothing.

NOTES: **Event_connect_error** is set to E_FAILURE if client is unreachable.

**Event_Send_Reply**

Send a reply **Item** to event interpreter identified by **ReplyID**.

INTERFACE:

```
void Event_Send_Reply(ReplyID, Item)
Event_EI_request_t Item;
Bee_gl_port ReplyID;
```

PARAMETERS:

| | |
|---|---|
| **Item** | Request to be sent. |
| **ReplyID** | ID - sent with request- on which event interpreter expects the reply from the client. |

RETURNS: Nothing.

NOTES: **Event_connect_error** is set to E_FAILURE if event interpreter is unreachable.

**Event_Receive_Reply**

Receive a reply **Item** from a client server request on port **Port**, size of received message is placed in **Size**.

INTERFACE:

```
void Event_Receive_Reply(Size, Port, Item)int Size;
Bee_gl_port Port;
Event_client_reply_ Item;
```

PARAMETERS:

| | |
|---|---|
| **Size** | Size of received message. |
| **Port** | Event interpreter port. |
| **Item** | Reply received from client. |

RETURNS: Nothing.

NOTES: **Event_connect_error** is set to E_FAILURE if client is unreachable.

**Event_Host_To_Ascii**

Convert event message from host format to Ascii format.

INTERFACE:

```
void Event_Host_To_Ascii(E, CE)
Event_t E;
Event_Ascii_t CE;
```

PARAMETERS:

| | |
|---|---|
| **E** | Original event message. |
| **CE** | Converted event message. |

RETURNS: Nothing.

NOTES: This function is automatically called by the `send` functions of the event protocol if the event interpreter is located on a heterogenous node.

**Event_Ascii_To_Host**

Convert event message from Ascii format into format used by host.

INTERFACE:

```
void Event_Ascii_To_Host(E, CE)
Event_Ascii_t CE;
Event_t E;
```

PARAMETERS:

| | |
|---|---|
| **E** | Original event message in Ascii format. |
| **CE** | Converted event message in host format. |

RETURNS: Nothing.

NOTES: This function is automatically called by the `receive` functions of the event protocol if the received message is in Ascii format.

# I. BEE Summary

## I.1 Event Sensors

### I.1.1 Language Independent Sensors

**Event_Sensor(Name, Class, Attr, Variant, EID)**
Event sensor for a user defined event class (with EID parameter).

### I.1.2 C Language Sensors

**EVENT_SENSOR(Name, Class, Attr, Variant, EID)**
Event sensor for a user defined event class (with EID parameter).

**EVENT(Name, Class, Attr, Variant)**
Event sensor for a user defined event class.

**BEGIN_PROCEDURE(Name)**
Activation of an event range of class E_PROCEDURE.

**END_PROCEDURE(Name)**
Termination of an event range of class E_PROCEDURE.

**BEGIN_EVENT(Name)**
Activation of an event range of class E_EVENT.

**END_EVENT(Name)**
Termination of an event range of class E_EVENT.

**POINT_EVENT(Sensor_Name)**
Event point of event class E_EVENT.

## I.2 Event Sensor Functions

**Event_Sensor_Max()**
Maximum number of event sensors allowed in client program.

**Event_Sensor_Control(C, Flag)**
Disable/enable all event sensors of a given event class C.

**Event_Sensor_Filter(C, Filter)**
Define global filter **Filter** to be applied to all event sensors of event class C.

## I.3 Event Initialization Functions

**Event_Initialize (C)**
Initialize the event processing facility for client C.

**Event_Finalize (C)**
To be called before the client finishes its execution on a node.

**Event_Create_Table (ET, ENT)**
Allocate an event table (system or task event table).

**Event_Cleanup_Table (ET, ENT)**
Deallocate a event table (system or task event table).

## I.4 Event Naming Functions

**Event_class_t Event_Register_Class(Name, Type, Size)**
Define a new event class **Name** with a variant part of type **Type** occupying **Size** bytes.

**Event_Client_Name()**
Return name of client program.

**Event_Class_Name(C)**
Return name of event class C.

**Event_Class_ID(Name)**
Return the event class descriptor of a predefined or user event class.

**Event_Register_Name(Name, EID)**
Enter name **Name** of event sensor into symbol table and return its event sensor descriptor **EID**.

**Event_Sensor_Name(EID)**
Return name of event sensor given an event sensor descriptor **EID**.

## I.5 Event Generator Functions

**void Event_Generate(Class, Attr, EID, Variant)**
Event_Generate assembles the components event sensor **EID**, class **Class**, attribute **Attr** and event class specific information **Variant** into an event message and passes it to the event handler.

**Event_Regenerate (E)**
Event_Regenerate passes an event message to the event handler without modifying the timestamp.

## I.6 Event Handler Functions

**Event_Handle (E)** Determine list of attached event interpreters for event **E** and pass `E` to their `Every` function.

**Event_Enable (C)** Enable the event class C in the current event table.

**Event_Delete(C)** Delete the event class C from the current event table.

**Event_Disable(C)** Disable the event class C.

## I.7 Event Interpreter Functions

**Event_Delete_Interpreter (EI_ID)**
Delete event interpreter **EI_ID** from the EI table.

**Event_Detach_Interpreter (EI_ID)**
Execute the `Final()` function of an event interpreter for all event classes it has been attached to and delete it from the current event table.

**Event_Disable_Interpreter (EI_ID)**
Disable event interpreter **EI_ID**.

**Event_Enable_Interpreter (EI_ID)**
Enable event interpreter **EI_ID**.

## I.8 Event Service Functions

**Event_Enter_Interpreter(S, I, E, F, Class, Filter, Enter_Mode, Client_Death_Mode)**
Enter event interpreter with service name N into name server domain.

**Event_Lookup_Interpreter(S, I, E, F, C, Filter, Location)**

Look up event interpreter at global name server.

**Event_Enter_Service(S, Enter_Mode, D)**
Make BEE service S known to the global name server in the domain b(D) and return its BEE port. **Enter_Mode** specifies what to do if the service already exists.

**Event_Lookup_Service(S, D)**
Lookup service S in the BEE name server domain D.

**Event_Attach_Client_Port(Port)**
Attach event interpreter to a client specified by its client server port **Port.**

## I.9 Event Interpreter Control Functions

**Event_Replay(F)** Read events from an event file **F.**

**Event_Update_Rate(Nr)**
Set the event interpreter update rate.

**Event_Update_Time(Time)**
Set maximum time between event interpreter window updates.

**Event_Display(N)** The name N of the workstation to be used by event interpreters for display.

**Event_Window(N, X, Y, W, H, T, L)**
Window configuration hints for event interpreter.

**Event_Font(F)** Name of type font for text.

**Event_Histogram_View(Title, Xname, Yname, Xtype, Ytype, Xlimit, Ylimit)**
Layout information for the histogram view.

**Event_Linegraph_View(Title, Xname, Yname, Xtype, Ytype, Xlimit, Sliding_Window, Ylimit)**
Layout information for the linegraph view.

## I.10 Miscellaneous Functions

**Event_Virtual_Client_Time(E)**
Convert the time stamp of client event **E** to event interpreter time.

**Event_Get_Time_Delta(E, Delta)**
Compute the difference between the timestamp in event E and the current clock.

**Event_Table_Inheritance(Flag)**
Enable/disable the inheritance mode for event tables.

**Event_Remote_Print (Flag)**
Redirect the output of PRINT_MSG macros to event interpreter **"bee_print_msg".**

**Event_Init_Debug_Switches()**
Set event kernel debug switches.

**Event_Version ()** Print BEE's version number.

**Event_Verbose (Flag)**
Enable/disable verbosity when executing event kernel functions.

**Event_Lookup_Retry (Retry)**
Set the number of retries when looking up an event interpreter service.

**Event_Buffer_Size (Nr_of_events)**
Size of event buffers allocated by the event kernel.

## I.11 Event Protocol Functions

**Event_Send_Init(Port, E)**
Send init message E reliably to event interpreter at port **Port**.

**Event_Send_Every(Port, E)**
Send every message E reliably to event interpreter at port **Port**.

**Event_Send_Final(Port, E)**
Send final message E reliably to event interpreter at port **Port**.

**Event_Send_Registration(Port, E)**
Send client registration message E reliably to event interpreter at port **Port**.

**Event_Send_Client_Death(Port, E)**
Send client death message E reliably to event interpreter at port **Port**.

**Event_Send_Descriptor(Port, E)**
Send type descriptor message E reliably to event interpreter at port **Port**.

**Event_Send_Request(Port, Request, ReplyPort)**
Send a **Request** to the client server at **Port**, waiting for an answer on **ReplyPort**.

**Event_Send_Reply(ReplyID, Item)**
Send a reply **Item** to event interpreter identified by **ReplyID**.

**Event_Receive_Msg(size, Port, E)**
Receive an event message **E** on port **Port**, report size of message in **Size**.

**Event_Receive_Request(Size, Port, Request, ReplyID)**
Receive a **Request** at port **Port**, size of received message is placed in **Size**.

**Event_Receive_Reply(Size, Port, Item)**
Receive a reply **Item** from a client server request on port **Port**, size of received message is placed in **Size**.

# References


[1] Ziya Aral and Ilya Gertner.
High-Level Debugging in Parasight.
In *Workshop on Parallel and Distributed Debugging*, pages 151-160. ACM, Madison Wisconsin, May, 1988.
Also published in SIGPLAN Notices, Volume 24, Number 1, January 1989.

[2] Emmanuel A. Arnould, Francois J. Bitz, Eric C. Cooper, H. T. Kung, Robert D. Sansom and Peter A. Steenkiste.
The Design of Nectar: A Network Backplane for Heterogeneous Multicomputers.
In *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 205-216. ACM/IEEE, Boston, April, 1989.

[3] Peter Bates.
Debugging Heterogeneous Distributed Systems Using Event-Based Models of Behavior.
In *Workshop on Parallel and Distributed Debugging*, pages 11-22. ACM, Madison Wisconsin, May, 1988.
Also published in SIGPLAN Notices, Volume 24, Number 1, January 1989.

[4] Thomas E. Bihari and Karsten Schwan.
*Dynamic Adaptation of Real-Time Software for Reliable Performance.*
Technical Report OSU-CISRC-5/88-TR17, Ohio State University, May, 1988.

[5] Bernd Bruegge and Peter Hibbard.
Generalized Path Expressions - A High Level Debugging Mechanism.
*Journal of Systems and Software* 3:265-276, 1983.

[6] Eric C. Cooper, Peter A. Steenkiste, Robert D. Sansom, and Brian D. Zill.
Protocol Implementation on the Nectar Communication Processor.
In *Proceedings of the SIGCOMM '90 Symposium on Communications Architectures and Protocols*, pages . ACM, Philadelphia, September, 1990.
Also published as CMU Technical Report CMU-CS-90-153.

[7] I.J.P Elshoff.
A Distributed Debugger for Amoeba.
In *Workshop on Parallel and Distributed Debugging*, pages 1-10. ACM, Madison Wisconsin, May, 1988.
Also published in SIGPLAN Notices, Volume 24, Number 1, January 1989.

[8] Riccardo Guesella and Stefano Zatti.
The Accuracy of the Clock Synchronization Achieved by TEMPO in Berkeley UNIX 4.2BSD.
*IEEE Transactions on Software Engineering* 15(7):847-853, July, 1989.

[9] Dieter Haban and Dieter Wibranietz.
A Hybrid Monitor for Behavior and Performance Analysis of Distributed Systems.
*IEEE Transactions on Software Engineering* 16(2):197-211, February, 1990.

[10] Jeffrey Joyce, Greg Lomow, Konrad Slind and Brian Unger.
Monitoring Distributed Programs.
*ACM Transactions on Computer Systems* 5(2):121-150, May, 1987.

[11] Michael J. Kaelbling and David M. Ogle.
Minimizing Monitoring Costs: Choosing between Tracing and Sampling.
In *23rd International Hawaii Conference on System Sciences*, pages 314-320. January, 1990.

[12] Ted Lehr, Zary Segall, Dalibor Vrsalovic, Eddie Caplan, Alan Chung, and Charles Fineman.
Visualizing Performance Debugging.
*IEEE Computer* 22(10):38-52, October, 1989.

[13] Allen D. Malony, Daniel A. Reed, David C. Rudolph.
Integrating Performance, Data Collection, Analysis and Visualization.
*Performance Instrumentation and Visualization.*
ACM Press, 1990, pages 73-97.
Edited by Margaret Simmons, Rebecca Koskela.

[14] Russell D. McLaren and William A. Rogers.
Instrumentation and Performance Monitoring of Distributed Systems.
In *Proceedings of the Fifth Distributed Memory Computing Conference*, pages 1180-1186. IEEE, April, 1990.

[15] Barton P. Miller, Morgan Clark, Jeff Hollingsworth, Steven Kierstead, Sek-See Lim, and Timothy Torzewski.
IPS-2: The Second Generation of a Parallel Program Measurement System.
*IEEE Transactions on Parallel and Distributed Systems* 1(2):206-217, April, 1990.

[16] David M. Ogle, Karsten Schwan, and Richard Snodgrass.
*The Dynamic Monitoring of Real-Time Distributed and Parallel Systems.*
Technical Report GIT-ICS-90/23, Georgia Institute of Technology, May, 1990.

[17] S. P. Reiss, E. Golin and R. Rubin.
Prototyping Graphical Languages with GARDEN.
In *IEEE Conference on Visual Languages*. IEEE, 1986.

[18] Richard Snodgrass.
A Relational Approach to Monitoring Complex Systems.
*ACM Transactions on Computer Systems* 6(2):157-196, May, 1988.

[19] David Socha, Mary L. Bailey and David Notkin.
Voyeur: Graphical Views of Parallel Programs.
In *Workshop on Parallel and Distributed Debugging*, pages 206-215. ACM, Madison Wisconsin, May, 1988.
Also published in SIGPLAN Notices, Volume 24, Number 1, January 1989.

[20] Peter Steenkiste.
Nectarine - A Nectar Interface.
1988.
Carnegie Mellon University, Internal document.

[21] H. Tokuda, M. Kotera, C. MErcer.
A Real-Time Monitor for a Distributed Real-Time Operating System.
In *Workshop on Parallel and Distributed Debugging*, pages 68-77. ACM, Madison Wisconsin, May, 1988.
Also published in SIGPLAN Notices, Volume 24, Number 1, January 1989.

[22] W.W.Wilcke, D.G.Shea, R.C.Booth, D.H.Brown, M.E.Giampapa, L.Huisman, G.R. Irwin, E.Ma, T.T.Murakami, F.T.Tong, P.R.Varker and D.J. Zukowski.
The IBM Victor Multiprocessor Project.
In *Proceedings of the First Conference on Hypercube Multiprocessors*. IEEE, Monterey, California, March, 1989.

# BEE Functionality