A Framework for Dynamic Program Analyzers

Bernd Bruegge, Tim Gottschalk and Bin Luo
Carnegie Mellon University
School of Computer Science
Pittsburgh, PA 15213

Abstract

BEE++ is an object-oriented application framework for the dynamic analysis of distributed programs. The main objective of BEE++ is to provide a common platform for monitoring and debugging. BEE++'s class library consists of a rich set of classes for event processing to support a variety of visualization, monitoring and debugging needs. It also provides for customizability of event processing through inheritance. Users can derive customized graphical debugging and visualization systems from a set of base classes. BEE++'s other design goals are the support of dynamic program analysis for distributed heterogenous target applications at runtime with predictable overhead. The design is based a symmetric peer-peer architecture, including the ability to dynamically configure target applications and monitoring tools. The dynamic analysis tools can be distributed across nodes, which provides significant performance gains for visualization applications. In addition, the framework can be instantiated for a variety of communication protocols.A TCP/IP based instance of the framework has been ported to several machine architectures including Sun, Vax and Cray-YMP. BEE++ is based on BEE[10], a portable platform for monitoring implemented in C but has been completely reengineered in C++ using the object-oriented design methodology OMT. Performance measurements indicate that the runtime overhead of the object-oriented version is not significant when compared with the C version.

This work was supported in part by Defense Advanced Research Projects Agency (DOD) monitored 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.

OOPSLA'93, pp. 65-82

1 Introduction

Dynamic program analysis, that is, performance and correctness debugging, is one of the least understood activities in software development and is practiced with the least amount of discipline. It is often approached with much hope and little planning. There is still much to be learned about debugging and, in particular, about what leads to successful debugging.

A major problem is still the inability to express the mismatch between the expected and the observed behavior on the level of abstraction maintained by the user. A general approach is to characterize a program's behavior (observable effects and interactions of components of a system) as a set of events. Event-based systems are attractive due to their simplicity. By viewing a program as an event-based system, a user can easily instrument it with event sensors to denote the occurrence of specific events. Whenever an event sensor is encountered, an event is generated and one or more tools connected to the system are notified. These tools are used individually or in concert to detect the desired correctness or performance information.

The effectiveness of event-based systems relies on two assumptions: First, the dynamic behavior of a system can be partitioned and observed with sensors. Second, it is possible to understand the system by viewing the sequence of events generated by these sensors with a set of program analysis tools.

The main drawback of an event-based model is that events are in many ways specific to the way the system is viewed, which is often related to a certain level of abstraction maintained by the system. For example, the process control part of an operating system would have create-process, suspend-process and resume-process as its most simple or primitive events. The same system with respect to its file I/O subsystem might

have open-file, create-file and close-file as primitive events. The problem is even more complicated though, because in many cases the interesting events are crossing several levels of abstraction, involving a composition of sensors from low-level as well as high-level abstractions.

One possible solution that has been proposed is to provide a small debugging platform with a small set of primitive events based on which the user can define composite events[3]. This is a powerful idea, but does not quite address the full problem. Any decision to include or exclude a certain event from the platform will cause problems, because it is easy to foresee debugging situations for which events cannot be specified because the primitive events are not part of the platform.

Another solution is to enlarge the platform and add a new primitive event every time we encounter the need for one. Unfortunately, the power of a debugging tool cannot not come from adding as many nuts and bolts to the platform as possible. The designer of the tool can simply not foresee all possible scenarios in which the tool is going to be applied. In addition, such a powerful tool can be quite overwhelming and might not be very helpful for the casual user which we believe is the main type of user for dynamic program analysis tools.

We believe the problem has more to do with the inflexibility in the formulation of primitive events. A user should be able to modify an existing primitive event with ease and we believe that this is one of the crucial elements of a successful debugger design. This kind of flexibility can be easily achieved through inheritance. Inheritance provides exactly what a debugger user desires: a new situation looks almost like another one, but only after a few modifications. We believe that a primitive set of events should be customizable based on inheritance. Furthermore, the tools which manipulate these event should be customizable as well.

2 The BEE++ Framework

A promising solution to the above problems is the use of an object-oriented application framework. A framework is a set of classes that embodies an abstract design for solutions to a family of related problems and supports reuse at a larger granularity than routines or classes[17].

In the last few years there has been a proliferation of frameworks to make the job of developing programs easier. The main goal of these frameworks is to speed up the software development process by providing reusable code. Some of these frameworks are simply class libraries covering general data structures such as collections, search tables or trees. The first such framework was the Smalltalk class library. The NIH library[13] is a reimplementation of the Smalltalk class library for the C++ programmer.

Other frameworks are designed to support the development of certain applications. An application framework is a framework that defines much of an application's standard user interface, behavior and operating environment. MacApp is an application framework that makes the task of building applications programs for the Macintosh faster and easier[21]. Examples of application frameworks to ease the building of graphical applications such as user interfaces are the Andrew toolkit (Atk) [26], the X Toolkit (Xt)[24], Interviews[20] and ET++[30].

BEE++ is an object-oriented application framework for the development of software-based distributed dynamic analysis tools. BEE++ is fundamentally an event processing system since it views the execution of a distributed program as a stream of events. Therefore, BEE++ provides customizability of all aspects of events processing: customizability of events, customizability of event views, and customizability of event configurations.

Event and view customizability is achieved through the inheritance mechanism available in C++, BEE++'s implementation language. BEE++'s entire event processing system is encapsulated in a set of core base classes. From these, additional classes needed to support various aspects of dynamic analysis tool development can be derived. Thus, we provide the user with a prepackaged hierarchy of classes and the means to create customized classes through derivation of the appropriate base class(es). While this assumes knowledge of object-oriented design and how subclassing is implemented in C++, it is the most nonintrusive way to extend or modify the functionality of BEE++, since the components of the event processing model can be altered or replaced without modifying existing code. This avoids the recursive problem of debugging a user-

customized analysis tool, in particular a debugger itself, because the user can assume that the class library is verified and validated.

Configuration customizability allows the dynamic establishment of user-defined events with user-defined views that can be modified during the execution. A well-designed event processing system should support three kinds of event analysis: post-game analysis, runtime monitoring, and during-game analysis. Post-game analysis, also called post-mortem analysis, is the traditional way to analyze event traces and many sophisticated tools have been developed for this type of monitoring [14], [19],[22]. However, post-mortem analysis is unacceptable for many problems, as we will examine below. Configuration customizability is necessary to achieve *runtime monitoring*, that is, the ability to monitor the behavior of an active system while it is running. Runtime monitoring is often the only way to monitor non-terminating programs such as servers. Runtime monitoring is also useful to monitor processes that take a long time to complete but deliver useful termination information much earlier during the execution. In many cases the behavior of an algorithm can be judged from monitoring information obtained quite early during the execution. An example where this is useful is the design of computationally expensive algorithms. In Section 8 we demonstrate the usefulness of runtime monitoring for algorithm design, that is, the exploration of algorithms to solve a certain problem, in this case the traveling salesman problem. One problem in connection with runtime monitoring is that the human users are hard-pressed to digest visualizations unfolding in real-time[14].

Thus we provide a special set of classes for during-game analysis, which allows a user interface to display past events while new events are still pouring in. This allows the user to browse a visual display of the system as it appeared back in time without losing new information. Our class abstractions hide many synchronization and caching schemes needed to implement this feature efficiently.

A naive implementation of runtime monitoring can place a high demand on system resources. To reduce the execution overhead, many implementations provide filters to suppress the generation and/or interpretation of unnecessary events [29]. The overhead can also be reduced by delaying the event interpretation to the postprocessing phase. Existing implementations that do provide runtime monitoring separate event generation and analysis by sending events to remote monitors, which may also combine event streams from different nodes[23][25]. Most of these systems, however, offer a rather static connection between client program and event processing system. Users are expected to specify queries statically before the execution or interpret the events afterwards[18].

BEE++ is a reincarnation of an earlier software development platform for program monitoring called BEE[6], implemented entirely in C, which supported customizability by providing a system of templates for clients and tools. However, in practice, this kind of customizability involved considerable reworking of existing code.

The paper is organized as follows. Section 3 provides a top level overview of BEE++ and discusses event processing architectures that can be built with the framework. Section 4 provides the rationale behind the selection of the classes developed during the analysis and design of the framework to support a single platform for monitoring, debugging and visualization tools. The complete framework is represented as a functional and object model in OMT[28] and we will therefore make extensive use of the OMT notation when discussing the rationale behind the abstractions and the services provided to framework users and other classes. Sections 5 and 6 explain the design goals and the detailed architecture. Section 7 describes object design decisions that improve the efficiency of BEE++. BEE++ has proven to be quite successful in debugging, visualizing and monitoring distributed applications. Section 8 demonstrates the usefulness of customizable event processing in combination with runtime monitoring for algorithmic design. Section 9 explains the result of experiments that illustrate the cost of event processing. Section 10 summarizes our experiences with BEE++ and provides ideas for further work.

3 An Overview of BEE++

Our functional model for event processing is composed of three processes: event generation, event interpretation, and the event transmission between generator and interpreter. For visualization applications, the target application performs the bulk of the event generation and the tool performs most of the event interpretation. On the other hand, a debugging

tool might generate program control and data events for interpretation by a target application. Hence, both target applications and monitoring tools perform event generation and event interpretation.

Note that while event generation and event interpretation can be combined in the same entity, the ability to abstract event processing from physical processes is a central feature of BEE++. Clearly, we allow users to move event interpretation out of the target application to another node. This not only makes it possible to do sophisticated runtime analysis with less impact on the application, but event streams of different parts of the application can also be combined to present a global picture of the application.

The top level architecture of any program under BEE++ can be described with a data flow model consisting of three main processes: the target program, the dynamic analysis tool and the event configuration manager as shown in Figure 1.

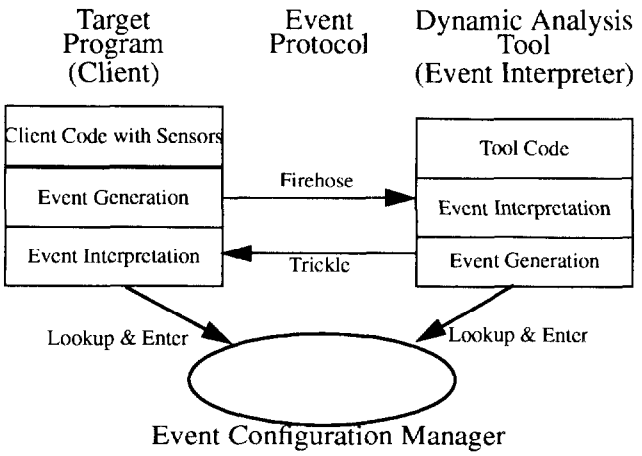


Figure 1 : BEE++'s top level system architecture

Events flow between the target program and the dynamic analysis tools over two distinct communication pathways which are called the *Firehose* and the *Trickle*, respectively. The Firehose consists of methods that send event instances to the dynamic program analyzer which processes them using user-supplied routines. The analyzer can also request information from the target program via the Trickle (We will discuss the Firehose and Trickle in more detail in Section 4.5).

The set of all the methods defined by the Firehose and the Trickle, that is, all methods of the class *event*, is called the *event protocol*. The event protocol defines the capability of the communication between clients and event interpreters. The chief reason the event protocol provides bidirectional traffic is to support performance debugging as well as correctness debugging. In monitoring and visualization situations, the target program performs the bulk of the event generation and sends the events to the tool for interpretation. In debugging situations, the dynamic analysis tool generates program control and data events and sends them for interpretation by the target application. The connection between the target application and the analyzer is provided by the event configuration manager and can be done at any time during the execution of the target application.

For the remainder of this paper we associate the term *client* with an instrumented target application and the term *event interpreter* (EI) with any dynamic program analyzer, which can be either a debugger, monitor or a visualizer. However, as we will discuss below, both, clients as well as event interpreters do quite a bit of event generation and event interpretation; this symmetric functionality distinguishes BEE++ from many other distributed dynamic analysis tools. Hence, we use the term *event processor* to denote either a client or an event interpreter.

3.1 Event Forwarding

The design of many dynamic analysis tools is centered around a client-server architecture. This is sufficient if the tool is either a debugger or a performance monitor, but not both. The goal of our framework was to provide a uniform platform for both activities and thus our architecture follows a symmetric, peer-to-peer approach. We believe that many attributes and methods can be shared by both the target application and the tool. For example, as shown in Figure 1, both event processing components are able to perform event generation as well as event interpretation. By deriving both target applications and analysis tools from a common base class, we believe we can achieve a symmetric architecture.

Consider the following thought: since BEE++ can be used on most any target application, it should be entirely possible to use BEE++ to analyze itself. That is, we should be able to treat an event interpreter as a candidate target application. Therefore, an event interpreter should be able to generate events exactly like the original target application. A key motivation for

this scenario is *event forwarding*. A target application might be connected only to a single event interpreter for client efficiency reasons. However, often multiple views of the execution are required, e.g. multiple event interpreters need to access the same event stream. This common performance bottleneck can be easily solved by event forwarding. The event interpreter, connected to a set of additional event interpreters, regenerates the event sent by the original target program. The event forwarding architecture scales quite well and reduces the bandwidth demands on the communication channel between target and first event interpreter.

To implement such a forwarding device, the entire BEE++ event processing model must be encapsulated into a set of classes such that both of the following methods are possible: (1) the forwarding device can be derived through multiple inheritance, and (2) the forwarding device can be formed by instantiating both the target application and event interpreter in the same address space. Hence, we use event forwarding as a key test of the integrity of the overall BEE++ class architecture. BEE++ distinguishes itself from other dynamic analysis frameworks in its ability to achieve this functionality via both methods.

4 The Object Model

To permit customizability of the entire BEE++ event processing model, we represent each component of the data flow model (client, event interpreter, event generation, event interpretation, event transmission, event configuration manager) by one or more classes. In the following we discuss the main classes of BEE++ as they relate to this model.

4.1 Class BeeProcessor

The commonality between target application and dynamic analysis tools, that is, the abstraction of an event processor is expressed in BEE++ by the BeeProcessor class which is shown as the center of the object model in Figure 2.

One of the key aspects of BEE++ is that all distributed processes running under BEE++ (monitors, debuggers, visualizers, event configuration manager, applications) are derived from this single base class. Every thread of control in BEE++, either in the same address space (local) or across a network, is represented by an instance of class BeeProcessor and all BEE++ methods are executed in the context of a BeeProcessor.

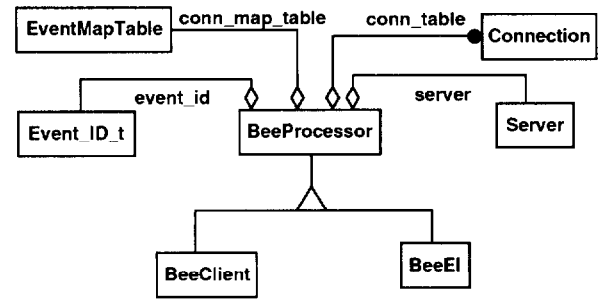


Figure 2 : Object model of the BeeProcessor

In fact, the BEE++ framework uses an internal BeeProcessor to bootstrap itself on start-up and shut things down after the last local BeeProcessor terminates. Thus, BEE++ is designed to support both multithreaded target applications and multithreaded analysis tools.

The BeeProcessor provides for the basic functionality needed by any target application or analysis tool: the ability to communicate with another BeeProcessor. However, while the BeeProcessor houses the entire event processing system (both event generation and event interpretation), it does not actually contain built-in event processing functionality. Rather, event processing methods are added through the derivation of new classes. In addition, the BeeProcessor does not have built-in communication capabilities; instead, it contains low-level communication objects that will be discussed in Section 6. On the other hand, the BeeProcessor does provide housekeeping details such as basic thread management, initialization and shutdown, command-line options, the ability to map communication objects to each other, and interactions with the event configuration manager. As shown in the object model, BeeProcessor relates to several other classes: The class Event_ID _t contains various information about this BeeProcessor. Server is an association to a Server object which allows it to serve connections over the Firehose or the Trickle.

The association conn_table points to a table of Connection objects; each remote BeeProcessor that this BeeProcessor is connected to (and the Server can communicate with) is represented by a Connection object. This association provides the ability for a single client to connect to multiple event interpreters and a single event interpreter to connect to multiple clients.

The conn_map_table association relates the BeeProcessor with the class EventMapTable; the *event map*

table allows a BeeProcessor to quickly determine the sender of an incoming event. It maps, via a hash table, the event identifiers of incoming events received by the Server to the appropriate Connection object in the conn_table, since each Connection object represents a sender.

The BeeProcessor handles many normal and unexpected program termination conditions so it can notify any remote BeeProcessors about its death and shutdown gracefully.

The BeeProcessor serves as a superclass for clients represented by the class BeeClient and event interpreters represented by the class BeeEI. In the following sections we discuss the BeeClient and BeeEI classes in more detail.

4.2 Event Generation: Class BeeClient

Although both target applications and analysis tools can perform all aspects of event processing, the target application usually performs the vast bulk of event generation. Therefore, we derive a class from BeeProcessor called BeeClient that is specially designed for event generation. Since the BeeProcessor has no event processing methods of its own, the BeeClient introduces methods for dealing with the Firehose and Trickle communication pathways. The BeeClient devotes its full attention to running an instrumented target application. Events from other event interpreters arrive infrequently, so instances of BeeClient typically employ asynchronous means to detect and interpret them.

Since target applications are external to BEE++, we use the class BeeClient as an abstract representation of the target application. Thus, the BeeClient deals with the special issues of instrumenting and executing a thread of target application code.

4.3 Class Sensor

Intuitively, we know some sort of association must exist between the target application code and the BeeClient. So far, we've used the notion of an "event sensor" to capture this association. We model this association as a class and hence introduce class Sensor.

A Sensor provides a placeholder for an event that is either user-defined or predefined by BEE++. When a Sensor is encountered, e.g. *triggered*, runtime data is loaded into the event and it is sent off to each analysis tool that has *bound* itself to that Sensor. A tool binds itself to a Sensor through the following process.

Each event within a Sensor contains type information about the data it contains. When an analysis tool first communicates with a BeeClient, it asks the BeeClient for type information about all the Sensors it knows about. The BeeClient tracks every Sensor it encounters during program execution in a sensor_table (see Figure 3). If the tool can understand the type of event within a particular Sensor, then it asks that Sensor to send it data whenever the Sensor is triggered. Note that for any Sensor encountered after a tool is connected to a BeeClient, a sensor initialization event is automatically sent to the bound tool announcing its presence.

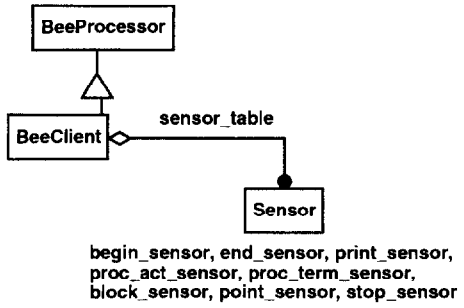


Figure 3 : Event sensor table

4.4 Event Interpretation: Class BeeEI

Since dynamic analysis tools perform the bulk of event interpretation, we associate them with the term event interpreter (EI). An event interpreter "interprets" events typically by displaying them in a graphical manner for further interpretation by a human user. Unlike the BeeClient class, which is designed largely for event generation, the dynamic model of an event interpreter is specially optimized for event interpretation: it spends much of its time just waiting for new events to arrive so they can be processed quickly.

From a system architecture point of view, dynamic analysis tools can be seen as interactive interfaces[28], that is, they are dominated by interactions with external agents, such as humans or other processes. External agents are independent of the system, so their inputs cannot be controlled by the dynamic analysis tool. Event interpreters, therefore, have to deal with two kinds of events: (1) events from a human

interacting with the system (e.g. GUI events) and (2) events from the target application (e.g. BEE++ events).

The event interpreters provided by the BEE++ framework form a hierarchy of classes. BEE++ provides users with a prepackaged set of event interpreter classes and the means to create customized classes through derivation of the appropriate base class(es). Figure 4 shows the BeeEI class hierarchy provided by BEE++.

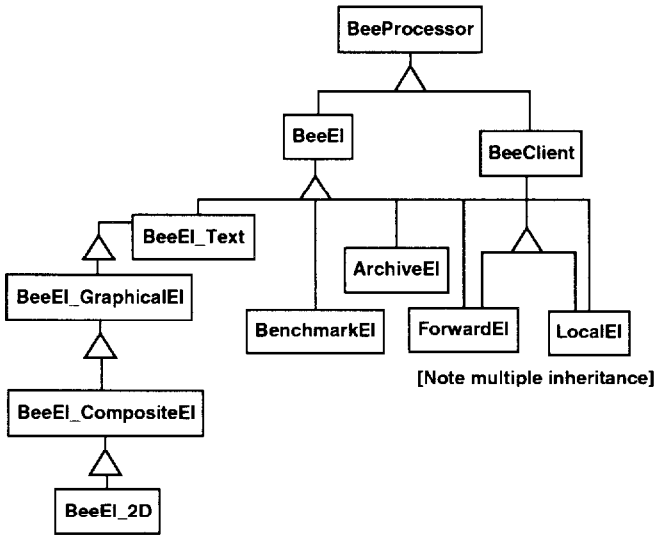


Figure 4 : BEE+'s event interpreter hierarchy

The BeeEI class is the base class for all BEE++ event interpreter classes. It provides only what is absolutely needed to build a minimal "monitoring" tool. It can fully process BEE++ events but has no means to display them.

The user can derive traditional text- or graphic- oriented tool classes from this class as well as classes where a user interface (or any screen output) would be inappropriate or unnecessary. In all these cases, the size and complexity of the code of the derived event interpreter is directly proportional to its functionality; an event interpreter directly derived from the BeeEI class does not carry any of the heavy baggage required for graphical visualization tools.

For example, the framework provides an ArchiveEI class that dumps events to a trace file. A user can also use the BeeEI class as a basis for interpreters that forward events to other interpreters. This is an interesting case of multiple inheritance because the ForwardEI provided by the framework is derived from BeeEI as well from BeeClient. In addition, a benchmarking class BenchmarkEI is provided. This is actually a very simple derived class with no additional methods. The main purpose of the BenchmarkEI is to provide an upper bound on the performance characteristics of an event interpreter in performance comparison with other event based platforms. This class is in fact used for the performance evaluation of BEE++ described in section 9.

The BeeEI_Text class deals only with events originating at the client and prints textual notification messages when the events arrive. It also adds some simple interactive prompting for the definition of client breakpoints and single-stepping at runtime (Single-stepping in the current BEE++ implementation means stepping from one BEE++ event sensor to another.)

The BeeEI_GraphicalEI class provides a graphical user interface based on the X Window System. It seamlessly integrates BEE++ event processing with X event processing and provides a simple menu bar and scrolling text region for messages.

The user interface design of this class and subsequent classes derived from BeeEI_GraphicalEI define an overall Motif-compliant layout policy while giving the tool developer the freedom to customize specifics such as menus, menu items, window arrangements etc. as needed.

The BeeEI_CompositeEI class adds additional user interface functionality that graphics-intensive tools might find beneficial, such as sensor count, event count, and event rate windows, and a scrolling text region for diagnostic messages. This class also provides a basic GUI for interactively controlling the execution of target applications for debugger support.

4.5 Event Class

BEE++'s event-processing model is based on the observation that dynamic understanding problems can not be dealt with a priori by program analysis tools that provide a set of predefined events. Assuming that BEE++'s library of existing events is found deficient, BEE++'s mechanism for event customizability allows a user to create a custom event and use it like any predefined event. To provide customizable events, we had two choices. We could provide a set of primitive events and a set of language constructs to compose higher level events. This is the approach taken, for example, by Generalized Path Expressions[5] and

EBBA[3]. Another solution is the specification of high level events from low level events by way of inheritance. This approach is supported by BEE++. We believe that it is easier to use for the casual user, because no new language concepts have to be learned to express new events.

We chose to model the association between targets and dynamic analysis tools as an object. Thus, we introduce class Event to serve as the base class for all BEE++ event classes. To support many kinds of BEE++ events, we derived a rich hierarchy of predefined event classes from class Event, which we will explore below.

Class Event contains two attributes, Family and Event_ID. The Family attribute (and other similar attributes of derived event classes) partitions the event space into equivalence classes. At each level of the Event class hierarchy, BEE++ supports predefined event classes such as the invocation of a procedure, as well as user-defined events to mark important milestones in the execution of an application. The Event_ID uniquely identifies the BeeProcessor that generated the event by storing its network node, OS-level process, OS-level thread, BeeProcessor id, and BeeProcessor instantiation.

To motivate how the class hierarchy was derived from Event, we examine how events are actually used in BEE++.

4.5.1 Firehose and Trickle

When designing the communication pathways between the target applications and the program analysis tools, we assumed different bandwidth requirements for the different directions of the event flow. Performance debugging and visualization are generally non-interactive and produce vast volumes of events flowing from the target application to the tool. An instrumented target program, for example, can have an extraordinarily high event rate if an event sensor is placed in a very tight loop. Interactive debugging produces a far smaller number of events; debugging events usually loop between the tool and the target application in the form of requests and replies. These events occur relatively infrequently as they are almost always generated in response to a user command. Note that we regard events generated by the program execution to be part of the former case. Therefore, correctness debugging requires a reliable but possibly slow protocol, whereas in performance debugging the protocol should provide a fast response, in some cases, can occasionally drop a packet. Note that since BEE++ analysis tools support during-game analysis, they cannot discard data; they can however chose to display only the latest information.

In light of these observations, we established two distinct communication pathways between any given target application and analysis tool, which as introduced earlier, are the Firehose and the Trickle. The Firehose is used for high bandwidth communication from target applications to tools. The Trickle is used relatively infrequently whenever a tool needs to communicate with the target application independent of the activity on the firehose. The top of the event class hierarchy provided by the framework is illustrated in Figure 5.

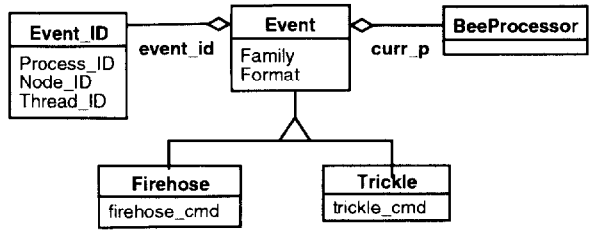


Figure 5 : BEE++'s event hierarchy

4.5.2 Class Firehose

From the Firehose, we establish an event hierarchy used in many instrumentation systems through the derivation of Init, Every, and Final event classes as shown in Figure 6. These classes are customized largely for dynamic program analysis as a whole; they define the bulk of BEE++ events.

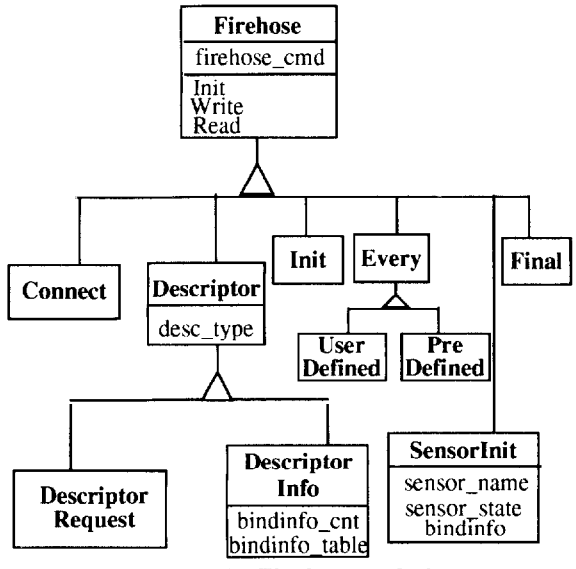


Figure 6 : Firehose subclasses

Since an event interpreter must dynamically bind to event sensors within a target application, we define a Descriptor event to facilitate the exchange of event type information. When an interpreter opens a connection, it sends the application a DescriptorInfo event containing a list of all event types it can or wants to recognize. The application then binds that interpreter to only those sensors that generate events of that type. In turn, these sensors generate SensorInit events to announce their presence to the interpreter.

The target program fires an Init event when it starts executing. Events of class Init are used for initialization of the analysis tool and are customized according to the functionality of the tool itself. For example, sophisticated GUI-based interpreters derive Init events with complex user interface attributes such as window size, title, placement.

Certainly the most important class is class Every. All events related to the execution of a program are derived from Every. We derive the UserDefinedEvery class to serve as the base class for all customizable Every events. For example, a scattergraph would derive an Every event from this class with two attributes (X and Y coordinates) whereas an Every event for a linegraph might only contain a single value. Creation of a user-defined event involves defining three methods: Init, Write, and Read. The Init method is invoked after event instantiation to set any internal attributes. The Write method is invoked by the sender (typically a sensor) to package the attributes of an Event into a byte stream suitable for transmission. An event may already maintain itself in such a way for efficiency reasons. Read does the opposite; it unpacks a byte stream into an Event again.

We also derive the PreDefinedEvery family of classes to provide common useful services, such as the ability to send text messages and important runtime information such as line numbers, function call invocations, and notification when breakpoints are encountered.

The Final event is sent upon death of the target program. In practice, we do not use the Final class since its functionality is largely supplanted by the Connect event class, whose derived classes specifically deal with the administration of a network connection.

4.5.3 Class Trickle

The Trickle and its derived event classes shown in Figure 7 are designed largely for asynchronous remote program control and asynchronous monitoring (e.g. "probing"). The Trickle event hierarchy is designed around a request-reply protocol. TrickleRequest events are subclassed into ClientConnect, BlockCont and Probe event classes. TrickleReply events are currently nothing more than acknowledgments, but can potentially support complex query interactions.

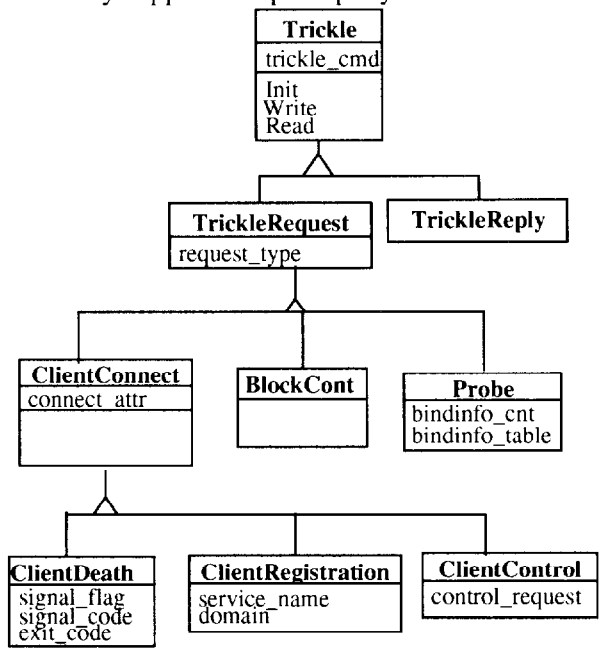


Figure 7 : Trickle subclasses

The ClientConnect class, which is very similar to the Connect class on the Firehose, handles the administration of an asynchronous connection. Chiefly, it provides the ability to connect to an already running target application. Thus the Trickle provides services similar to the ptrace facility in Unix to control the thread/process and read/ write data. In contrast to the standard ptrace, which assumes a static connection between client and debugger, a Trickle can be dynamically established which allows BEE++ to support two kinds of event sessions: an unplanned event session, where the client runs for a while before connecting to an event interpreter, and a planned event session where the event interpreter connects to the client before the latter starts with its execution.

From the ClientConnect class, we derive ClientDeath, ClientRegistration and ClientControl classes that specifically support remote program control, such

as the ability to pause, restart, or kill the application. The BlockCont event is sent by the event interpreter to continue the execution of a blocked target application. The Probe event is very similar to a DescriptorInfo event on the Firehose. An interpreter fills the attributes of this event with the Every event types it wishes to receive and then sends it to an application. The application then triggers those sensors that can generate those events, independent of the execution of the program. The BEE++ event kernel ensures atomicity of event generation.

5 System Design

BEE++'s class library was designed to support a variety of architectures ranging from single processes to parallel and networked programs. Special consideration was given to the goal of supporting a variety of heterogenous architectures in a wide area network; it is almost inevitable that a network becomes heterogenous even if it is initially configured as a set of homogenous nodes. With the advance of high-speed networks, new protocols are being developed that utilize the increasing bandwidth better than existing protocols and the design allows to move the BEE++ framework easily to the new protocol. The separation of event generation and interpretation allows the developer to write client programs and event interpreters in different languages employing different compilers, runtime systems and operating systems.

BEE++ achieves heterogeneous event processing by converting all events into a canonical form before sending them across the Firehose or Trickle. The conversion to canonical form is described by a set of encoding rules which define exactly how data is arranged as a sequence of bytes when it is sent over the network. There are two main classes of encoding rules: primitive and complex. BEE++'s primitive encoding rules describe how ambiguous atomic data types are represented. For example, BEE++ represents integers as four-byte quantities using Big-Endian byte ordering (the MSB is in the lowest memory position). String buffers are left untouched since their representations are unambiguous. In practice, BEE++ event instances contain several fields of data. While each field can be encoded using primitive encoding rules, the ordering of each field is still ambiguous. Therefore, complex encoding rules describe the ordering of the event fields. The encoding approach used by BEE++ is similar to Sun's Remote Procedure Call (RPC) mechanism which uses a set of primitive and complex encoding rules called the External Data Representation (XDR). BEE++ has also adopted XDR's representation for four-byte integers.

The Firehose as well as the Trickle make use of a set of encoding methods provided by the superclass Event. However, because the C++ language and the underlying Sun RPC mechanism used in the implementation of the encoding methods do not support the communication of classes over the network, we implemented an internal scheme to track the type information of event instances. Each event stores its event type in the attribute Event_ID_t.

Event-based systems based on BEE++ can span different network protocols. BEE++ supports TCP/IP for use with standard Ethernet and a customized protocol for use with Nectar, an experimental fiber-optic network developed at Carnegie Mellon University[1]. Since BEE++ can instantiate multiple client and server objects in the same process, it can support multihomed hosts; hence, in the future it may be able to act as a gateway.

5.1 System Topologies

The system architectural design issues for BEE++ included both, the network architecture connecting the client program with the event processors as well as the event processor architecture, that is, the distribution of the event processors. BEE++ provides the user with the ability to set up event configurations, where one or more clients are connected to one or more event interpreters. Event configurations enable the user to distribute the monitoring activities across the network to decrease the impact of the monitoring process on any single processor.

One of the main features of BEE++ is its flexibility with respect to its system topology. Users need to be able to customize the monitoring process by connecting client programs and dynamic analysis tools and rearranging the connections at runtime. BEE++'s dynamic model supports this requirement in several ways: The event interpreter can interact with more than one client during a single monitoring session. Alternatively, a client can interact with more than one event interpreters during a single monitoring session.

BEE++ supports a wide spectrum of topologies. The simplest topology is a local event interpreter (called LocalEI in Figure 4) that uses the same address space as the client program. Generation of events means the invocation of procedures within the event interpreter at the appropriate locations. Local event interpreters are ideal when information can be accumulated during program execution, and displayed at the end. They are less well-suited for runtime monitoring; displaying status information or updating the screen inside an event interpreter can slow down the client significantly.

BEE++ also allows users to group event interpreters or clients together as needed. Several clients executing simultaneously (either as separate threads or processes) can be attached to an event interpreter as shown in Figure 8.

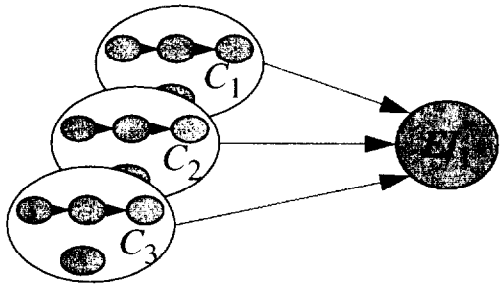


Figure 8 : Monitoring multiple clients

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 monitoring and for dynamic load balancing purposes.

Figure 9 shows a client connected to several event interpreters, each of them tapping on the same event stream, but providing different views of the behavior. The different event interpreters can either be on the same

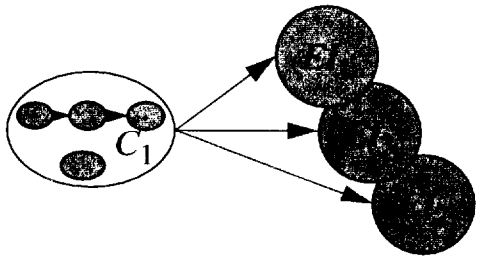


Figure 9 : Multiple view monitoring

node or on different nodes. The client overhead can be reduced by having the client send the events to just one event interpreter which then forwards events to other interpreters. This configuration can easily be achieved using BEE++.

5.2 Boundary Conditions

BEE++ supports two ways of setting up a connection between an application and an analysis tool. In an unplanned connection, an application starts up first and then event interpreters connect during execution. In a planned connection, the event interpreter is started up first followed by the start of the application. Planned connections support the dynamic model employed by many existing debuggers. Unplanned connections are useful for monitoring daemons or long running applications that are exhibiting performance problems but cannot be restarted easily.

6 BEE++'s Network Architecture

In order for BEE++ to provide a flexible symmetric communication model, we deliberately did not provide the event processor e.g. class BeeProcessor with any built-in means of communications. Rather, we packaged the entire communication mechanism into a simple set of classes isolated entirely from the rest of the system. Thus, a BeeEI or BeeClient can instantiate the desired communication protocol at runtime. Likewise, BEE++ can be ported to new protocols by simply deriving new classes. Any necessary customization is achieved through inheritance and overriding. The classes providing BEE++'s communication model are organized into the following three hierarchical layers:

- The *Port layer* is the lowest layer of BEE++ and provides the basic means to send/receive data. The classes in this layer encapsulate the implementation of the event protocol on top of an existing network protocol.
- The *Connection-Server layer* provides a protocol-independent means of communication for servers and connection classes that are connected via the Port class.
- The *Firehose-Trickle layer* provides the event protocol to be used by the BeeClient and BeeEI classes. This layer does not introduce any new classes but instantiates classes from the previous layer to establish Firehose and Trickle communication pathways.

At first glance at this hierarchy it would appear that BEE++'s communication model follows a traditional client-server approach. However, BEE++ allows cli-

ent and server objects to be placed in the same entity, thereby providing peer-peer functionality. In the following we describe each of these layers in more detail to demonstrate the portability of the BEE++ framework and to show how the framework classes defined above can communicate with each other in a heterogenous network.

6.1 The Port Layer

The Port layer contains objects of class Port, its subclasses and the Event Configuration Manager class. Instances of class Port interact with each other and the Event Configuration Manager to provide the simplest form of client-server communication.

6.1.1 Class Port

The Port class provides a standard way of developing new communication primitives across different protocols and networks. It defines standard methods for opening and closing connections (OpenConnection, CloseConnection) as well as sending and receiving bytes of data (Send, Receive). In practice, Ports are first subclassed into the network protocol (e.g. Class TCP_IP or Class RMP) and then into the client-server relationship (TCP_IP_Client, TCP_IP_Server, RMP_-Client, RMP_Server etc.). Client and server Ports may be further broken down according to the type of server being used: iterative or concurrent. An iterative server Port handles only one client at a time, whereas a concurrent server Port can maintain many open connections to clients at once.

Client Ports need to know certain information about server Ports before they can open a connection. Therefore, all Ports have a method called GetPortInfo that returns all the network-dependent information needed for a client Port to connect to a server Port. Typically, a server Port's GetPortInfo method is called and the resulting string is stored in the event configuration manager. A client obtains this string from the event configuration manager and then calls the client Port's SetServerPortInfo method with this information. Then, the client calls the OpenConnection method on the client port and the connection is established automatically.

6.1.2 Class Event Configuration Manager

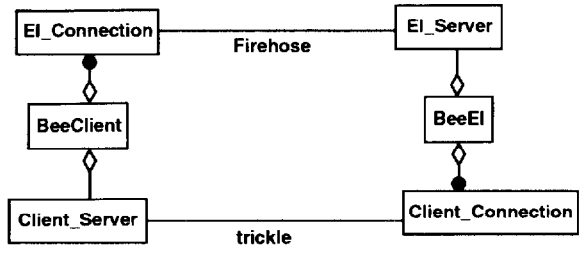
BEE++ has a special class for tracking event configurations called the Event Configuration Manager. This class has to be instantiated before any event configuration can be built. It allows event interpreters to connect to clients and vice-versa.

The Event Configuration Manager class manages the entire set of event interpreters and clients involved in a single monitoring session. Event configurations are dynamic, that is the set can grow as new clients/ interpreters are added at runtime or shrink as clients/ interpreters are stopped or die.

The Event Configuration Manager provides basically the functionality of a multiprotocol nameserver. The class has methods for entering, looking up, and removing services that are associated with the Ports described above. A service is the name associated with any operational server Port. Server Ports can be associated with both, target applications and event interpreters, and thus clients and event interpreters might provide the same service. The name space of the Event Configuration Manager is therefore partitioned into domains to distinguish between them. For example, clients belong to the CLIENT domain and event interpreters belong to the EI domain. Domains can be further partitioned into groups for system maintenance purposes.

6.2 The Connection-Server Layer

The Connection-Server layer introduces the classes Connection and Server as well as their subclasses. Connections contact Servers to establish a portable, protocol-independent means of communication. The model of communication is identical to the Port layer in that Servers register themselves with the Event Configuration Manager and are the contacted by Connections, e.g. clients, connect to them.



6.2.1 Class Server

The Server class contains a server-related Port member. We use the term *service* to describe the Server object in a program. A server contains methods for opening and closing connections (Open, Close) and handling information (PutMessage, GetMessage). A

Server stores information about itself with the event configuration manager so objects of type Connection can communicate with it. The Terminate method requests that the event configuration manager remove information about that service before shutting down the Server. The InstallHandler method allows the Server to be used in an interrupt-driven manner so a program does not have to poll for new events.

6.2.2 Class Connection

The Connection class contains a client-related Port member and provides the same services as the Server class. Like a Server, it contains the methods Open, Close, PutMessage, and GetMessage. In addition, it provides a Connect method which asks the event configuration manager for information about a remote Server and then establishes a communication path with it (the client Port connects to the server Port using the event configuration manager information as described above).

The Disconnect method shuts down the path created by Connect. The Connection class is subclassed into Client_Connection (for use with target applications) and EI_Connection (for use with monitoring tools). The Client_Connection also performs special duties related to buffering messages (event collections). The EI_Connection tracks special information about the target application as a whole, such as the procedure call stack and all BEE++ instrumentation sensors. In the future, it may even contain a copy of the target application's symbol table.

6.3 The Firehose-Trickle Layer

Up until now, the BEE++ layers have treated communication information as an untyped stream of bytes. However, the Firehose-Trickle layer communicates strictly with instances of class Event or its subclasses. The Firehose-Trickle layer is built on top of the Connection-Server layer and providing the highest communication layer in the BEE++ framework; it is used by clients and event interpreters to communicate with each other. Recall that methods for dealing with the Firehose or Trickle are not provided with the BeeProcessor class, but rather defined by the BeeClient and BeeEI classes. These methods are named GetFirehoseEvent, PutFirehoseEvent, GetTrickleEvent, and PutTrickleEvent. The BeeClient uses each Connection to communicate via a Firehose to a Server located in a BeeEI. Likewise, a BeeEI uses each Connection to communicate via the Trickle to a Server located in a BeeClient.

During initialization, the Firehose is used to send setup information in both directions. The Trickle allows the event interpreter to communicate with the client (usually asynchronously) independent of Firehose activity. The Firehose-Trickle methods Attach and Detach are called to connect or disconnect a Connection with a Server. As described earlier, the term service means any Server object and its associated BeeProcessor as a whole. Hence, the phrase "attaching to a remote service" describes either an event interpreter connecting with a running application (unplanned) or an application connecting with a running event interpreter (planned).

The illustration in Figure 10 shows how the different classes and subclasses of Port, Connection, Server, and BeeProcessor are instantiated to realize a specific event configuration (Ports are shown as gray circles).

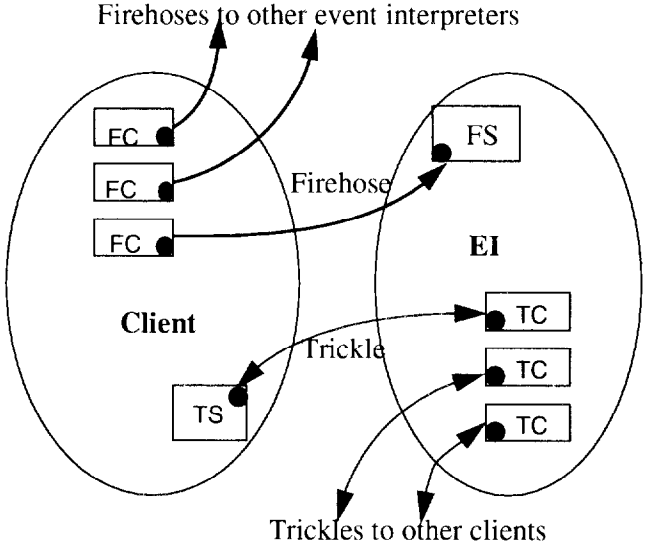


Figure 10 Instance diagram for an event configuration with 3 clients and 3 event interpreters

7 Efficiency

Event-based systems, no matter how carefully designed, introduce an overhead in the client execution. An important goal of any event based monitoring system is to minimize the effects of client perturbation

due to instrumentation and monitoring. However, in many cases the perturbation is acceptable, in particular if the insights in the program's behavior and the eventual speedup gained from the measurements improve the program significantly. BEE++'s design goal was therefore to provide an event processing system with *predictable* instrumentation cost. Several optimization mechanisms available in the framework allow to reduce the event overhead in those cases where the event overhead is too large. The two most important ones are event buffering and poll-driven instrumentation. Sometimes it is not possible to determine the overhead a priori. BEE++ supports the interactive setting of parameters that influence the event rate at run-time.

7.1 Event Probes

An event interpreter can communicate with a client in *probing* mode, meaning that it does not receive any sensor-generated events over the Firehose unless it specifically asks for them by sending a probe event over the Trickle. When this happens, the client forces that sensor to regenerate an event using attributes from the latest trigger action.

This can be a significant performance boost during long computations where the client only needs to be checked occasionally to ensure that nothing is wrong.

7.2 Event Collections

An *event collection* is a set of Firehose events that are collected on the client's side into an *event collection buffer* and sent to the event interpreter only when the buffer is full. Similar to blocked I/O, a large event buffer reduces the event overhead on the client side but increases the event latency. In one extreme case all generated events are collected and sent to the event interpreter after the client finishes execution. A familiar example is postmortem debugging. True runtime monitoring means that once a sensor is encountered, it sends events immediately over the Firehose.

BEE++ provides individual event collection buffers and buffers arguments for each connection between client and event interpreters. This allows a client to attach a debugger listening for a breakpoint while also being attached to a frequency counter.

Because event collections impact the event latency they can interfere with any real-time constraints placed on the event interpreter such as response time. BEE++ provides a mechanism that lets the user to flush the collection buffer at specific times. This allows interactive BEE++ users to choose between the conflicting goals of event efficiency and runtime monitoring.

8 Examples of "BEE++ Instantiations"

BEE++ has proven to be useful in several distributed applications for performance debugging, visualization and monitoring on Nectar, including NOODLES, a solid modeling application[11], the COSMOS switch-level circuit simulator[10], a chemical flow-sheet application and environmental simulations [9] and Mistral-3, a parallel solid modeling program based on an octree decomposition of modelling space[16]. BEE++ has also been interfaced to the visualization system Paragraph[14] by supporting the interpretation of PICL[12] events.

Another application for BEE++ is the visualization of algorithmic design. The screen snapshot in Figure

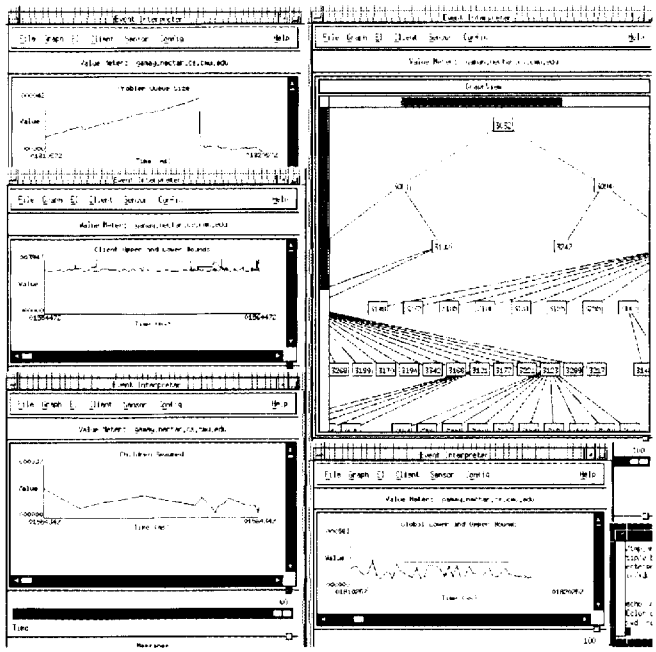


Figure 11 : Runtime visualization of a distributed traveling salesman problem

11 shows a "cockpit" involving several event interpreters to visualize the performance of a distributed Traveling Salesman Problem program using a client/server architecture[27]. The instrumentation of the application program with customized sensors and the development of the BEE++ cockpit was done by an application programmer in a few hours. Four load

meters and a customized event interpreter are used for the runtime visualization as follows. The upper left event interpreter shows the size of the queue of subproblems maintained by the server. The middle left window displays the local upper & lower bounds for subtours investigated by each client. The lower left window show the number of subtours (children) spawned by each client. The middle bottom window visualizes the development of the global upper&lower bounds over time. The window called GraphView is a customized graphical event interpreter showing a branch and bound tree as it is being searched during the execution. This event interpreter was built by subclassing the class BeeEI_2D from BeeEI_CompositeEI (See Figure 4 in Section 4.4) and combining it with existing tree viewing code. Runtime monitoring in combination with these views turns out to be extremely useful for this problem of algorithm design, because the developer can see the quality of the convergence of the algorithm without having to run the application to its completion.

9 Performance Evaluation

A bee collects the nectar without hurting the flower
Chinese Proverb

The performance of any event processing system depends critically on the application and its instrumentation. Currently there is no benchmark for event processing systems that covers a wide spectrum of applications. Given the lack of a good benchmark we used the event configuration described in [6]: An instrumented client executing a loop generating nothing but events and an event interpreter "processing" these events with an empty method.

To compare the performance of the C++ and C implementations, respectively, we conducted a set of 3 experiments measuring the event overhead of three event configurations: a 1-1 configuration consisting of a single client connected to a single event interpreter, a multiple monitoring configuration of up to four clients connected to a single event interpreter and a multiple view configuration of up to four event interpreters connected to a single client. As event interpreter(s) we used the BenchmarkEI from the class hierarchy described in Figure 4.

The performance is characterized in terms of three parameters: the event rate, the event overhead experienced by the client and the event latency. The *event overhead* is the runtime overhead per event sensor when executing an instrumented client program attached to an empty local event interpreter. The *event rate* is the number of events generated per time unit by an instrumented client program attached to an event interpreter. The *event latency* measures the time from when an event sensor is encountered to the point when the event is passed to a event interpreter routine for analysis. Event latency is an important metric to characterize the real-time capability of the event system[1].

The measurements were performed on a network of

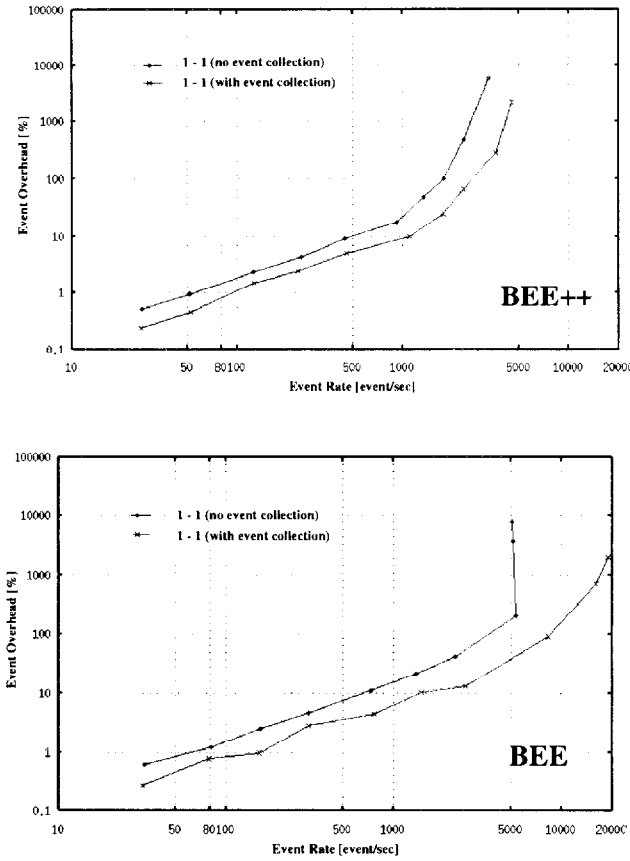


Figure 12 : Comparison of event overhead between BEE and BEE++

Sun4/330 workstations connected via Ethernet. The event protocol was instantiated with TCP/IP. All programs including event libraries were compiled with the same optimization switches and the measurements

1. Measuring these parameters in controlled experiments poses several problems described in more detail in[7].

were done on an otherwise "empty" network, that is, without the presence of other users.

The event latency was the same for BEE and BEE++ and was measured to be 20 msecs. The event overhead and event rates for the basic event configuration are shown in Figure 12. With a user tolerating an event overhead of 1%, BEE++ can generate 50 events/sec and 500 events incur an event overhead of 10%. Note that event collection buffering increases the event rates to 80 and 800 in the 1% and 10% cases, respectively. For comparison, the C implementation delivers 80 and 800 events/sec for unbuffered event transmission for the 1% and 10% data points. Thus, according to our measurements, the BEE++ user is paying a slight runtime performance penalty.

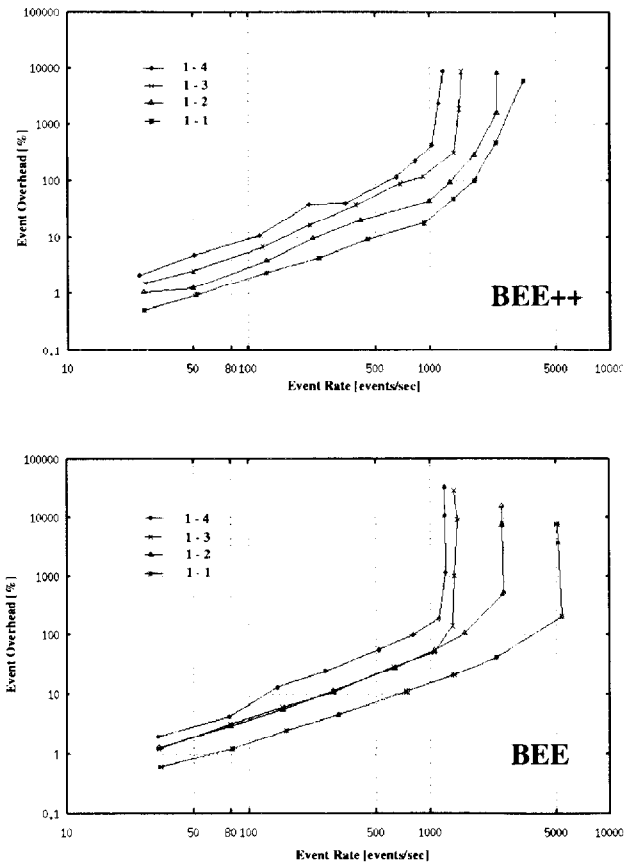


Figure 13 : Event overhead for multiple view monitoring

Figure 13 compares the event overhead for multiple view monitoring for event configurations of a client connected to up to four event interpreters and Figure 14 shows the event overhead caused by a single event interpreter connected to up four clients. In both cases, the user experiences a performance penalty in the C++ implementation, caused mostly by the additional procedure calls due to the use of inheritance in the event generation and event interpretation, respectively.

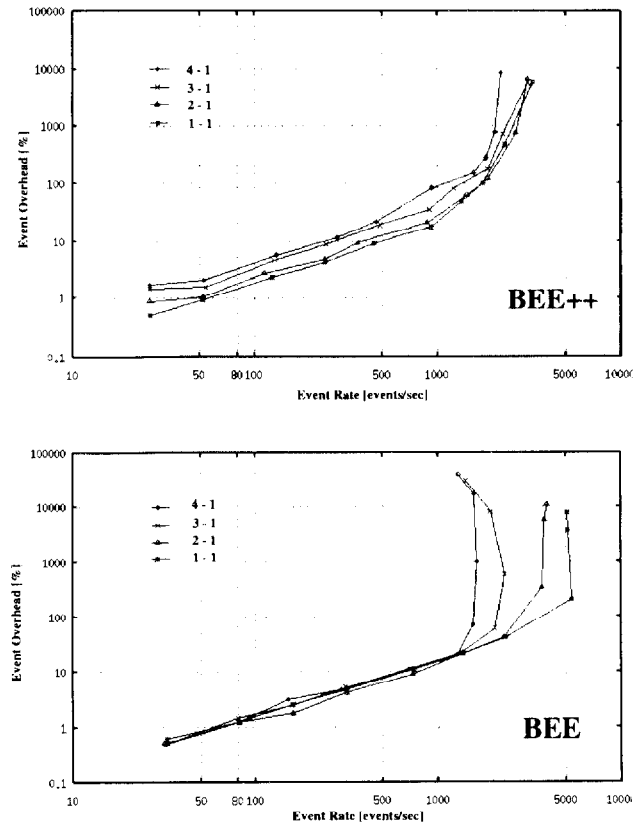


Figure 14 : Event overhead for monitoring multiple clients

10 Conclusion

In this paper we have described BEE++, a framework for distributed event processing systems providing user customizable instrumentation and visualization facilities based on inheritance.

BEE++ programs can run on a variety of architectures including Sun workstations under SunOS and Mach, Dec Workstations under Ultrix and on Cray supercomputers under UNICOS. The framework is written in C++ and X11R4/Motif 1.1. The event protocol is implemented on two network protocols, TCP/IP and Nectar RMP, and is currently ported to PVM[4]. The event kernel consumes less than 80K of executable code and supports the instrumentation of C and C++ programs. Compared with the C implementation,

the size of the overall executables shrunk by about 80%, most of it due to the use of inheritance

BEE++ has been used for the performance and correctness debugging of several distributed applications and as a result a sizable collection of BEE++ event interpreters has been created that is now available to new users. These "default" event interpreters range from distributed time profilers, linegraphs, scattergraphs and frequency counters to 3-d visualization tools.

The current version of BEE++ is available via ftp for interested users. We are also planning to continue our work on BEE++, in particular to improve the support for distributed debugging. The creation of sensors is currently the task of the application programmer, who has to insert the sensors in the application program which must be compiled before the execution starts. There are ways to relieve the programmer from this chore, for example, by introducing a mechanism that allows the dynamic creation of sensors at runtime. The dynamic creation of event sensors would also support the setting of breakpoints at runtime which is currently not supported by BEE++.

Finally, we are investigating a SymbolTable class that provides full access to the symbol table of a target program. BEE++ currently accesses only the symbols associated with event sensors. To support the full debugging paradigm, BEE++ must provide access to the symbol table associated with the application.

11 References

[1] E. Arnould, F. Bitz, E. Cooper, H.T. Kung, R. Sansom, P. Steenkiste, The Design of Nectar: A Network Backplane for Heterogenous Multicomputers, Proceedings of the 3rd Int. Conf. on Architectural Support for Programming, Languages and Operating Systems, Boston, 205-216, April 1989.

[2] P. Bates, Distributed Debugging Tools for Heterogeneous Distributed Systems, 8th International Conference on Distributed Computer Systems, IEEE Computer Society, San Jose, CA, 1988.

[3] P. Bates, Debugging Programs Using Event-based Models of Behavior, In Proceedings of the Workshop on Parallel and Distributed Debugging, pages 68-77. ACM, Madison Wisconsin, May, 1988. Also in SIGPLAN Notices, 24(1), January 1989.

[4] A. Beguelin, J. Dongarra, A. Geist, V. Sunderam, Visualization and Debugging in a Heterogenous Environment, Computer, 26(6), 88-95, June 1993.

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

[6] B. Bruegge, BEE: A Basis for Distributed Event Environments (Reference Manual), CMU-CS-90-180, Carnegie-Mellon University, November 1990.

[7] B. Bruegge, A Portable Platform for Distributed Event Environments. In Proceedings of the ACM/ ONR Workshop on Parallel and Distributed Debugging, 184-193. ACM, December, 1991.

[8] B. Bruegge, P. Steenkiste. Supporting the Development of Network Programs. In International Conference on Distributed Computing Systems, Texas. IEEE, May, 1991.

[9] B. Bruegge, H. Nishikawa, P. Steenkiste. Computing over Networks: An Illustrated Example. In 6th Distributed Memory Computing Conference, Portland. April, 1991.

[10] R. Bryant, D. Beatty, K. Brace, K. Cho, and T. Sheffler. COSMOS: A Compiled Simulator for MOS Circuits. In Proceedings of the Design Automation Conference, 9-16, ACM/IEEE, June, 1987.

[11] Y. Choi. Vertex-based Boundary Representation of Non-Manifold Geometric Models. Ph.D. thesis, Carnegie Mellon University, 1989.

[12] G.A. Geist et. al., PICL: A Portable Instrumented Communication Library, Reference Manual, Technical Report ORNL/TM-11130, Oak Ridge National Lab, Oak Ridge, 1990

[13] K Gorlen, S. Orlow, P. Plexico. *Data Abstraction and Object-Oriented Programming in C++*, Chichester, England, John Wiley & Sons Ltd., 1990.

[14] M. Heath, J. Etheridge, Visualizing the Performance of Parallel Programs, IEEE Software, 29-40, September 1991.

[15] D. Heller, Motif Programming Manual, Vol 6 in the X Series, O'Reilly & Associates, Inc. 1991

[16] N. Holliman, C. Wang and P. Dew. Mistral-3 : Parallel Solid Modelling. Technical Report TR 91-4, University of Leeds, January, 1991.

[17] R. E. Johnson, Brian Foote, Designing reusable classes, Journal of Object-Oriented Programming Vol 1 No 2, 22-35, 1988.

[18] R. LeBlanc and A. Robbins. Event-driven monitoring of distributed programs. Proceedings of the

5th Conference on Distributed Systems, IEEE 1985.

[19] T. Lehr, Z. Segall, D. Vrsalovic, E. Caplan, A. Chung, and C. Fineman. Visualizing Performance Debugging. IEEE Computer 22(10):38-52, October, 1989.

[20] M.A. Linton, J.M. Vlissides, P.R. Calder, Composing User interfaces with Interviews, Computer, Vol 22, No 2, 8-22, February 1989.

[21] Macapp 2.0 General Reference Manual, Apple Computer 1990.

[22] A. Malony, D. Hammerslag, D. Jablonowski, Traceview: A Trace Visualization Tool, IEEE Software, 19-28, September 1991.

[23] B. P. Miller, M. Clark, J. Hollingsworth, S. Kierstead, S. Lim, and T. 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.

[24] A. Nye and T. O'Reilly, X Toolkit Intrinsics Programming Manual, Vol 4 in the X Series, O'Reilly & Associates, Inc. January 1990.

[25] D. Ogle, K. Schwan, and R. Snodgrass. The Dynamic Monitoring of Real-Time Distributed and Parallel Systems. Technical Report GIT-ICS-90/23, Georgia Institute of Technology, May, 1990

[26] A. Palay, W. Hansen, M. Kazar, M. Sherman, M. Wadlow, T. Neuendorffer, Z. Stern, M. Bader and T. Peters, The Andrew Toolkit: An Overview, Proc. of the USENIX Technical Conference, 1988.

[27] J. Pekny, D. Miller, A. Kudva, An Exact Algorithm for Resource Constrained Sequencing With Application to Production Scheduling Under an Aggregate Deadline, Computers and Chemical Engineering, 17(7), 671-682, 1993.

[28] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, W. Lorensen, Object-Oriented Modeling and Design, Prentice Hall, 1991.

[29] R. Snodgrass. A Relational Approach to Monitoring Complex Systems. ACM Transactions on Computer Systems 6(2):157-196, May, 1988. Performance Evaluation

[30] A. Weinand, E. Gamma, R. Marty, Design and Implementation of ET++, a Seamless Object-Oriented Application Framework., Structured Programming Vol 10, No 2, 63-87, 1989.