ENHANCEMENTS TO THE GLOBAL EVENT DETECTOR TO IMPROVE FUNCTIONALITY AND PERFORMANCE

By

GAURI SUKHATANKAR

A THESIS PRESENTED TO THE GRADUATE SCHOOL
OF THE UNIVERSITY OF FLORIDA IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE

UNIVERSITY OF FLORIDA

1999

To my family

ACKNOWLEDGMENTS

First I would like to express my sincere gratitude to Dr. Sharma Chakravarthy, for giving me an opportunity to work on this interesting topic and for providing me with great guidance and support through the course of this research work. I am also thankful to Dr. Eric Hanson and Dr. Joachim Hammer for serving on my committee.

I would like to express my special thanks to Sharon Grant and for maintaining a well administered research environment and being so helpful in times of need. I am grateful to Hyoungjin Kim and Shiby Thomas for their invaluable help and fruitful discussions during the design and implementation of this work. Also, I would like to thank all my friends for their constant support and encouragement.

I would like to thank the Office of Naval Research, the Navy Command, Control Ocean Surveillance Center RDT&E Division, and National Science Foundation for supporting this work.

Last, but not the least, I thank my family for their endless love. Without their support, this work would not have been possible.

TABLE OF CONTENTS

Page

LIST OF FIGURES

Abstract of Thesis Presented to the Graduate School
of the University of Florida in Partial Fulfillment of the
Requirements for the Degree of Master of Science

ENHANCEMENTS TO THE GLOBAL EVENT DETECTION SERVER
TO IMPROVE FUNCTIONALITY AND PERFORMANCE

By

Gauri Sukhatankar
May 1999

Chairman: Sharma Chakravarthy
Major Department: Computer and Information Science and Engineering

Sentinel is an active object oriented database management system (DBMS) that monitors conditions associated with events and allows the specification of event-based rules. Events include method, temporal and external events. When an event is triggered, the condition associated with that event is evaluated and if it evaluates to true the action is executed.

The global event detection server (GED) has been designed to facilitate Sentinel applications to define rules on events occurring outside of their address space. The GED has a communication architecture consisting of sockets and remote procedure calls (RPCs) for passing external events across applications. The GED monitors external events using the producer consumer paradigm. In the distributed application environment it is important to accommodate various kinds of failures: consumer failure, producer failure as well as GED failure. To accommodate for failures, the GED persists events and

supports recovery. To accommodate for very high rates of event generation by the producer clients, buffer management was added to the GED.

Although the GED monitors external events and is recoverable, currently it cannot support multiple clients efficiently. Its performance needs to be improved in order to accommodate multiple producers and consumers exchanging a large number of events at any given time. Currently the GED passively passes events across applications without any computation on them. To facilitate “intelligent” forwarding and “filtering” the GED needs to be able to check some conditions before passing events. An expressive GED server will be one that has the capability to support Event Condition Action (ECA) rules on global events. The focus of this thesis is to make the GED scalable and improve its performance, and to extend the GED functionality to support ECA rules on global events.

CHAPTER 1
INTRODUCTION

An active database monitors the database state and reacts spontaneously when predefined events occur. Functionally, an active DBMS allows specification of event-based ECA rules and monitors the conditions associated with events. Events include domain specific events such as insert, update, delete for relational databases, method invocations for object oriented databases and temporal and external events. An object oriented database system supports user defined data types and provides powerful capabilities to model and persist complex objects. A number of emerging applications, such as computer aided manufacturing, power distribution network, automated office workflow control need to continually monitor their database state and quickly respond with proper actions to certain events. When an event occurs, the condition function is evaluated and if it evaluates to true, the action is executed. Thus, an active database supports applications with ECA rules. This active capability is useful in a single system. For applications that access more than one database we need to extend the active capability to work in a distributed environment.

In order to support active capability within applications, the local event detection server was designed. The LED allowed an application to define rules on local events. The GED was designed to extend the event specification capability so that an application can specify rules on events external to its address space. In order to support global (external) event specification and detection in a distributed environment, global event definitions

were added to SNOOP (an event specification language of Sentinel). The LED was also extended to support for communication with the GED and was called ELED (extended LED). The Global event detector (GED) was implemented to detect events that span multiple applications. Client applications connect to the GED server and register events with it to use its capability of global event notification. When external events are sent to the GED, it sends event notifications to consumer clients over a socket and the clients make remote procedure calls to send or receive events from the GED.

Although the GED monitored events in distributed applications, reliability issues had not been addressed in the initial design. In order to address robustness and recovery issues, reliable event detection and prorogation was needed. Hence, a recoverable GED was implemented that gets to a consistent state following various types of client failures and can also continue to provide services in a normal fashion when the GED itself recovers from system failures. The GED uses write-ahead logging for persistence as well as recovery. Buffer management is used to support flexible allocation and use of memory by the GED, as unlimited memory availability for storing events is not realistic. Events are stored in an event file (one per consumer) and in a buffer (whose size is specified in the GED configuration file) from where they are received by a consumer.

1.1 Motivation

The current work on the Global Event Detector supports event monitoring and recovery in a distributed application environment. However, it does not address the issues of scalability. When multiple clients connect to the GED and exchange a large number of events, performance of the GED must not deteriorate. Since the current implementation uses synchronous RPCs, a client process making a request to the server blocks till the

reply is returned. If multiple clients are making requests to the GED at the same time then a lengthy request by one client will cause the other clients to wait. Moreover, response time for clients will increase as more and more clients connect to the GED. To reduce these delays the GED should be able to handle multiple client requests concurrently. Hence the GED server needs to be a multitasking server. However, multitasking will involve concurrent access to shared data structures, which may cause race conditions. Hence these shared data structures also need to be protected.

The current implementation of write-ahead logging and buffer management has not been done in an efficient way. Each log record contains an event occurrence and its parameters and has a unique LSN, which is its log sequence number. Reading of a specific log record, given its LSN, involves reading the entire log file from the start to the position of the record LSN. This takes considerable I/O time, especially when the log file is large. Knowing offset of BLSN (LSN of last event in the buffer) will allow one to read an event without sequential read of the file. Storing the log in binary format instead of ASCII will make reads and writes faster.

In the current implementation the GED is just a passive mediator of events across different applications. In a typical situation the GED needs to have the capability to do filtering, delaying, or useful computation on the data associated with received events before sending events to the consumer. This means that the GED needs the capability of defining rules where useful actions can be taken under certain conditions on events arriving at the GED. A rule editor is currently being used in Sentinel to allow the user to specify rules on local events within each application. To support specification of rules on global events, the existing rule editor interface needs to be extended. A rule editor server

also needs to be designed that will do operations such as compilation, persistence and retrieval on the rules specified through the interface. The GED must support dynamic loading of these persisted rules at runtime so that rules will be fired when global events are notified to the GED. The focus of this thesis is to make the GED a multithreaded server to handle client requests concurrently, to provide efficient logging and buffer management and to extend the GED functionality to support ECA rules on global events.

The remainder of this thesis is organized as follows: Chapter 2 presents an overview of the event specification language and current support for rules in Sentinel. Chapter 3 discusses some design issues, analyzes their advantages and disadvantages, and describes how they are applied in the multithreading of the GED. In Chapter 4, we discuss more implementation details of the GED. Chapter 5 gives the performance measurements on the GED. Chapter 6 discusses design issues for supporting rules in the GED and Chapter 7 discusses implementation of rule support. Chapter 8 gives a conclusion along with discussion of the limitations and future work.

CHAPTER 2
OVERVIEW OF SENTINEL

Sentinel is an active OODBMS with an integrated approach. It supports primitive event detection and nested transactions as part of its kernel. In addition, it supports composite event detection and rule management as separate modules. Snoop [1] is an event specification language used in Sentinel for specifying ECA rules. It defines event and rule specification, supports event operators and parameter contexts. SPP is the preprocessor for SNOOP.

2.1 Types of Events

Four types of events can be identified in a distributed system:

Local primitive event. Any method of any object class is a potential primitive event. A local event is one which is predefined in the application. Primitive events include database events and temporal events. **Database events** correspond to database operations such as insert, delete or a method invocation on an object. Events are further refined into *primitive* events by using event modifiers, begin or end. For example, every instance of a *Stock* class may have a method *Insert***.** Then *Stock* can potentially invoke its *Insert* method and produce an event, such as *begin-of Insert*. **Temporal events** include absolute or relative temporal events. An absolute temporal event is specified as an absolute value of time such as a time string using the format <(hh/mm/ss)mm/dd/yy>. A relative temporal event corresponds to a unique point on the time line but in this case both the reference point and offset are explicitly specified A relative temporal event is

given as follows:

Primitive_event ::= event *event_modifier* method_signature

Local Composite Event. Local composite events are formed by applying a set of operators to local primitive events and local composite events. If E1 and E2 are either primitive or composite events then a local composite event is given as follows:

Composite_event ::= E1 operator E2

Global Primitive Event. Global primitive events are events that are either local primitive or local composite in one application and are referenced/used by another application. Since a global primitive event occurs outside of the application that uses it, there must be some way to define the event as well as to detect and transmit the occurrence. *App_name*, *Remote_event_name* and *Host_name* are attributes that solve this problem. *App_name* is the name of the remote application or its ID. *Remote_event_name* is the name of the event defined locally within the remote application. *Host_name* is the name of the host on which the remote application is running.

Global_primitive_event ::= *Remote_event_name::Host_name__App_name*

Example: E1 ::= produce::rain__prod

Global Composite event**.** If an event is composed of one or more events where at least one of them is a global event then it is called global composite event.

Example: compE ::= E1 operator E2

(at least one of E1/E2 is a global event)

2.2 Parameter Contexts

Snoop supports the notion of parameter contexts [2] to capture application semantics for computing parameters or consuming event occurrences (of composite

events). These contexts are precisely defined using the notion of initiator and terminator events. An initiator of a composite event is a constituent event that can start the detection of the composite event whereas a terminator is a constituent event that can detect the occurrence of the composite event. A composite event may be comprised of several primitive events that can be initiators and terminators of another composite events. When the occurrences of several primitive events constitute a composite event, different possibilities for detecting composite events exist. By carefully analyzing several classes of applications, four parameter contexts are proposed in Snoop:

1. Recent: In this context, only the most recent occurrence of an initiator for any event is used. All other occurrences of a constituent event will not be used in detecting another composite events and they will be deleted when the event started by the initiator occurs. The initiator of the event will continue to initiate another event until another initiator occurs.
2. Continuous: Each occurrence of the initiator of an event continuously initiates the event. A terminator may terminate one or more occurrences of the same event.
3. Chronicle: The initiator and terminator pair of a composite event is unique and deleted after the event occurred. They are paired based on chronological basis.
4. Cumulative: In this context, for each constituent event, all occurrences of the event are accumulated until the composite event is detected. In other words, the parameters of a composite event include the parameters of all the occurrences of each constituent event. All the occurrences of each constituent event are flushed whenever its associated composite event is detected. Detailed discussion of the parameter contexts is given in [3].

2.3 Event Operators

Figure 1 shows the types of events in Sentinel. The following is the summary of Snoop operators with brief explanations:

1. AND: Conjunction of two events, namely E1 and E2. The order of occurrence of E1 and E2 is irrelevant. example: E_AND ::= E1 ^ E2
2. OR: Disjunction of two events, namely E1 and E2, occurs when either E1 or E2 occurs. Example:E_OR ::= E1 || E2.
3. SEQ: Sequence of two events, namely E1 and E2. Occurs when E2 occurs after the occurrence of E1. Example:E_SEQ ::= E1>>E2.
4. NOT:Negation operator detects non-occurrence of an event, namely E2, in the closed interval formed by two events, E1 and E3.

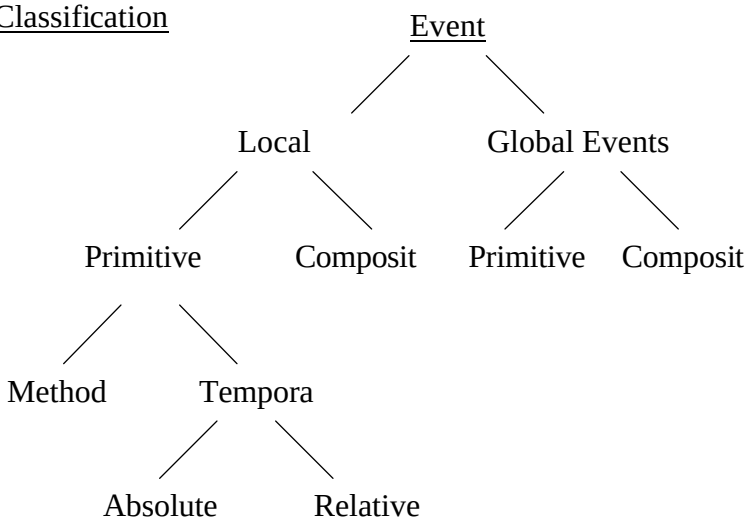
   Example: E_NOT ::= ¬ E2[E1,E3].
5. A: Aperiodic event is detected for every occurrence of E2 during the half-open interval formed by E1 and E3. Example: E_A ::= A(E1,E2,E3)
6. A*: Aperiodic closure event is a cumulative variant of the A operator. It is detected when E3 occurs provided E1 has already occurred. The occurrences of E2 are accumulated during the half-open interval formed by E1 and E3.

   Example: E_A_STAR ::= A*(E1,E2,E3)
7. P: Periodic event is detected for every time period specified by E2 during the half-open interval (E1,E3]. E2 is a time specification.

   Example: E_P ::= P(E1,E2,E3).

8. P*: Periodic closure event is a cumulative variant of P operator. A P* event, where E2 is a relative temporal event, is detected only once when E3 occurs provided the E1 has already occurred. Example: E_P_STAR::= P*(E1,E2,E3)

Event Classification



Event Operators

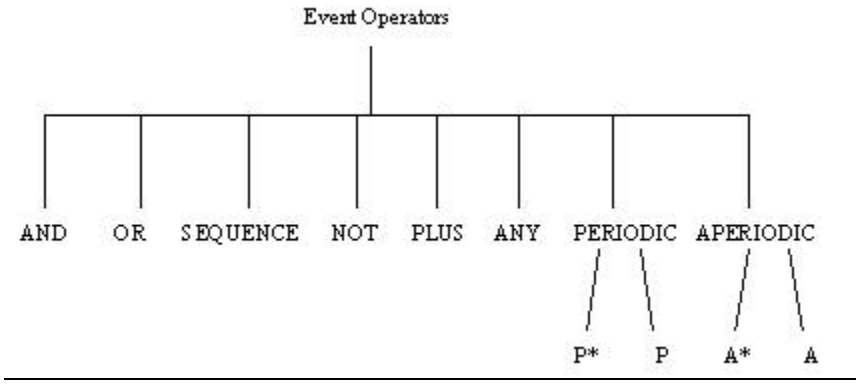


Figure 1: Types of Events in Sentinel

2.4 Summary of Event Detectors

In an object oriented DBMS, database events correspond to the execution of

methods of a class. Therefore, there must be a mechanism to trap the invocation of (or return from) a method when an event is signaled. In a centralized system the Local Event Detector (LED) [2] is used for detecting local primitive events and composite events within applications. In a distributed computing system, events across distributed applications need to be monitored. To accommodate global event detection, some extensions were made to the LED. The new LED is called ELED. Global Event Detector (GED) [4] is responsible for monitoring events from different applications in a distributed environment. It recognizes the occurrence of events, collects and records their parameters, and passes them to the interested applications where the rule managers trigger the action of ECA rules.

2.4.1 Local Event Detector

The purpose of the local event detector (LED) is to detect the occurrence of local primitive and local composite events within an application. The LED is implemented as a

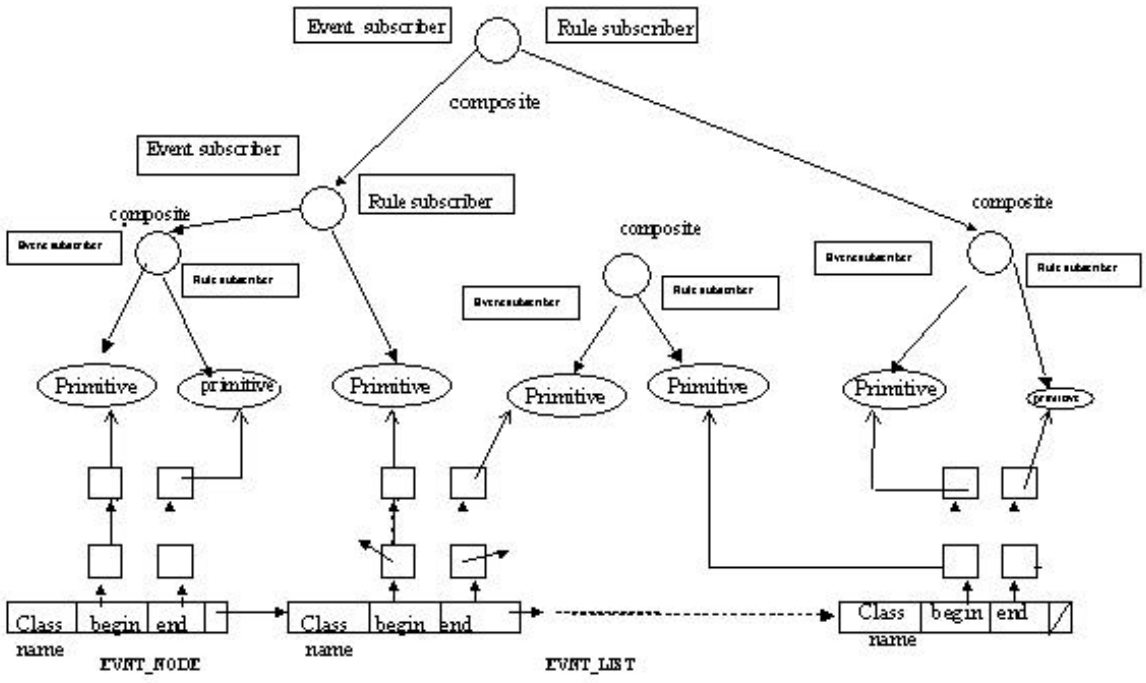


Figure 2: LED Data Structure

event, that is a leaf node of the event graph. The leaf nodes of the event tree graph correspond to primitive events from which the composite events are constructed. Each node of the event graph has an event subscriber and a rule subscriber that records the related composite events and rules. Whenever a primitive event is raised, it will notify its subscribers which are its parent nodes. The parent nodes (composite events) will maintain the occurrence of its constituent events as a part of its parameter lists, which are stored separately for each context relevant to the node. If the composite event occurs by the current notification, it is detected and notified to its subscribers. The parameter list is recomputed to include the new occurrences. For details of the LED, refer to [2].

In order to support rules on global events in the LED, the LED interface was enhanced to communicate with the GED This extended version is called ELED. In addition to detecting local events, the ELED will have to send an event notification to the GED server when an event is raised that is needed by other remote sites. Moreover, the ELED will detect a global event only when it receives an event notification from the GED server. To accommodate global events, a REMOTE class has been added to the class hierarchy in the LED. Similar to the LED, the ELED is an instance of the EVNT_LIST class that records information of all the event instances. Each node of the EVNT_LIST is related to a unique application and contains all the global event instances that are detected outside of this application. Each node contains a list (ELIST) of REMOTE nodes that become the leaves of the event graph. Whenever a global event is detected outside of the application, the GED interface will receive the event notification along with application ID and event parameter list from the server and further notify the

ELED. ELED then determines the specific EVNT_LIST node according to the application ID and propagates the event notification to its corresponding REMOTE event instance. According to its event subscribers and rule subscribers, a notified REMOTE event instance will further notify related composite events, that is its parent nodes.

2.4.2 Global Event Detector

The global event detector detects events that span several applications in a distributed application environment. Its purpose is to allow an application to detect events occurring not only at a local site, but also at other remote sites. It recognizes the occurrence of events, collects and records their parameters, and passes it to application rule managers where an ECA rule will be triggered.

Since each application has its own local event detector, a global event detector is responsible for detecting events that are defined at a remote site. Therefore, the global event detector (GED) adopts the client/server model and must be able to communicate with local event detectors at remote sites through RPC and socket-based communication. Figure 3 shows the GED communication architecture. Detailed discussion of the GED architecture alternatives is in [4].

First, a client process makes a socket connection to register with the GED server, and the server will record the socket address of this client and events that need to be detected by the GED if this client is a consumer. Event name list (*cname_l*) will be sent to the corresponding producer who generates the requested event, and the *GED_forward_flag* will be set to 1 with corresponding event in *cname_l.* Then, whenever the LED detects an event, it checks its *GED_forward_flag* to see if this event needs to be sent to the GED server. Global primitive events are first detected by the local

event detectors at their corresponding remote sites. Then, event notifications are sent to the GED server. When an event is notified to the GED server, it sends a message to each consumer on a socket. After the consumer has received this message, it makes a remote procedure call to the server and pulls the event (with its parameter list) from the server. Finally, the consumer traverses its ELED graph to propagate the global event.

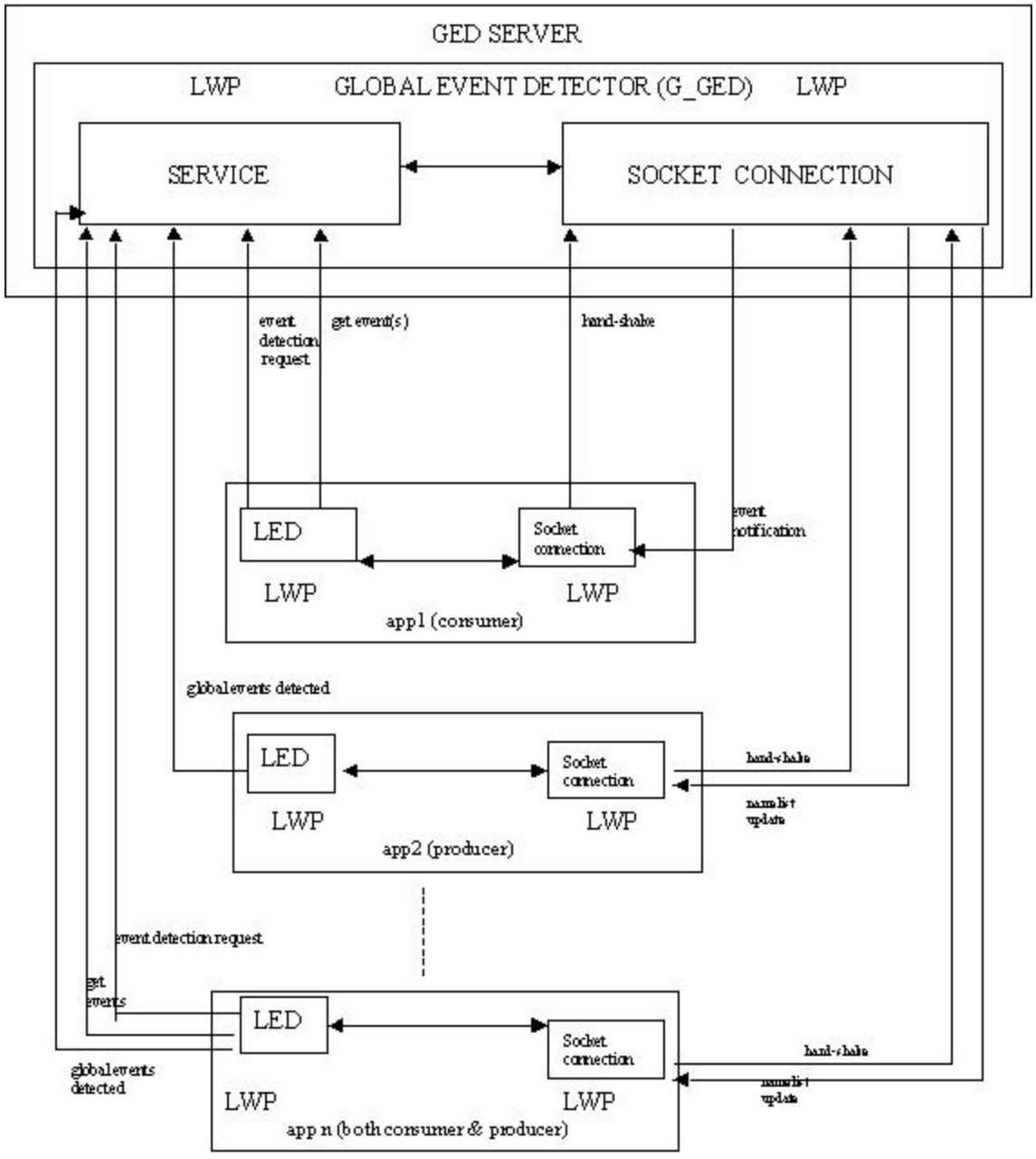


Figure 3: GED Communication Architecture

2.4.3 Global Event Graph

A PRIMITIVE class in the LED specifies primitive event objects; however, it is not appropriate to use this terminology in the GED since global primitive events denotes external events that are detected outside of the local application. Therefore, the GLOBAL class was introduced instead. GLOBAL class stands for the global primitive event objects. There are three attributes in GLOBAL class: *send_sname, send_ename, and event_no*. *send_sname* indicates the (consumer) application ID (machine name__application name) that is to be notified by the server after this event is raised. *send_ename* is the name of this event that application *send_sname* uses. It has the same value of *ename* attribute of a REMOTE class instance which is related to this global primitive event in application *send_sname*. *Event_no* denotes the instance number of the occurrence of this event.

Whenever a client registers with the GED server, it must send some information to the GED server to build the global event graph (G_GED) if global events are defined in the client. The information that a client sends is obtained from the global event specification file that is generated by the *sPP*. Refer to [4] for more information on the global event specification file. Similar to the LED, the G_GED is also an instance of EVNT_LIST. However, the NOTIFIABLE class that each ELIST points to is a GLOBAL class instead of PRIMITIVE or a REMOTE. So, when a global event is notified to the GED server, it traverses the G_GED and further notifies related composite events (parent nodes), and computes its parameter list. Figure 4 shows the data structure of the GED global event graph.

Global events are detected on the server using the event graph. An event tree is

created for each composite event and these trees are merged to form an event graph for detecting a set of composite events. This will avoid the detection of common sub-events multiple times thereby reducing storage requirements. For each node in the event graph, there is an event subscriber linked containing all the composite events that use this event as its constituent one. An event node has a pointer to its subscriber which becomes its parent node. Whenever a global primitive event is detected, it will propagate the event notification to its subscribers, that is its parent nodes. Event occurrences flow upwards as in a data-flow computation. The parent nodes maintain the occurrences of its constituent events along with their parameter lists which are stored for each context set to the node. If the composite event occurs by the last notification, it is detected and further propagates to its subscribers. Each time an event is raised, it will send this event to a specific application that had subscribed for the event.

2.5 Support for Rules in Sentinel

A sentinel application has the capability of defining Event Condition Action rules. The following paragraphs give a brief summary of the rule support in sentinel applications.

In the context of Sentinel, a rule consists of an event, a condition function of boolean type, action function of void type, and a few attributes. Once an event is detected by the system, the associated condition function is evaluated and the associated action function will be executed based on the result of the evaluation. Sentinel supports nested rules. When a rule's action raises an event that triggers rules there is a nested execution

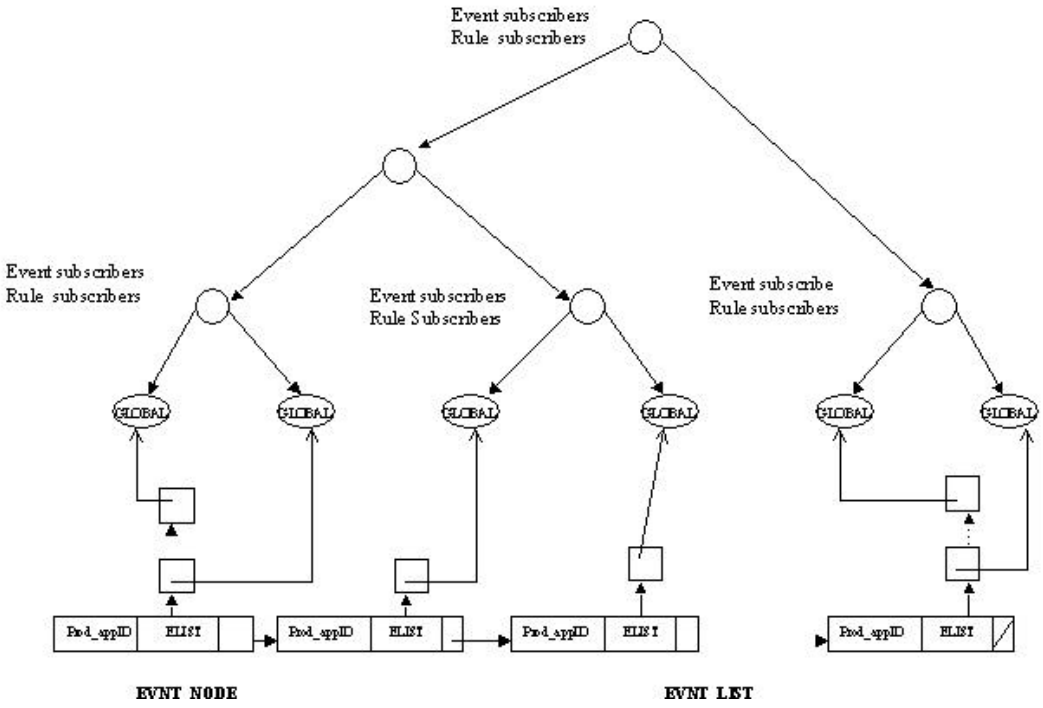


Figure 4: Global Event Graph

of rules. Detailed discussion of rule semantics is in [5]. As part of the rule semantics, it is necessary to know when to execute condition and action functions after the associated event has been raised. This is an issue about the *coupling* between the event and the condition-action pair. Currently, *immediate* and *deferred* coupling modes are supported between event and condition-action pair.

Sentinel supports multiple rules. An event can trigger several rules. Therefore, it is necessary to support rule execution mode that supports concurrent and prioritized serial execution of rules. Sentinel uses *Priority* classes for specifying rule priority. An arbitrary number of priority classes can be defined. A rule is assigned to a priority class by

indicating its number or the name of the class. Sentinel provides a global conflict resolution mechanism among the priority classes and concurrent execution of rules that belong to the same priority class.

Sentinel also introduces two types of rule *trigger* modes for specifying the time from which event occurrences are to be considered for the rule. The two trigger modes are *now* (start detecting all constituent events starting from this time instant) and *previous* (all component events since the event was detected last are acceptable). *Now* is the default trigger mode. Following is an example of a rule defined within an application on a global primitive event g1:

```
event g1 = STOCK_e1::manatee__app1;
rule gr1[g1, cond1, test_act1, RECENT];
```

The rules supported by Sentinel are specified in the classes (class-level) and the code (instance-level) of an application. They are referred to as internal (static) rules. Internal rules are specified in the application and processed. The application source code is processed by the Sentinel preprocessor (sPP), whose output is then processed by the Open OODB preprocessor and finally compiled by a C++ compiler into an executable. Rules on events within an application can also be specified externally by the user by using a rule editor. The condition and action functions specified by the user through the rule editor are compiled and rule information is persisted into the database using OQL queries [6]. Function names of the condition and action are persisted with the rule information into the database and the compiled function object code is made into DLLs. At application runtime, the *load_dyn_rules* routine (available as part of sentinel code) is called which uses the condition and action names to retrieve the function pointers by

using the dynamic loader utilities, *dlopen* and *dlsym*. In this way externally defined rules can be loaded into applications at runtime. Details of the rule editing and loading are in [7]. Whenever an event is detected within an application the condition function will be executed and if it evaluates to true the action function for the rule will be executed.

CHAPTER 3
DESIGN ISSUES FOR MULTITHREADING

This chapter discusses our approach to making the GED multithreaded. In addition to multithreading, several performance related improvements have been incorporated into the GED code.

3.1 Design Goals

The GED server was designed to monitor events in a distributed application environment where the client applications are producers and/or consumers of events. Consumer clients make RPC calls to register events with the server. Similarly producer clients make RPC calls to send instances of events to the GED. Consumer clients also make RPC calls to receive event occurrences when the GED notifies them. An RPC call is synchronous and blocking [8]. The mechanism of synchronous RPC is shown in figure 5. This means that a client making an RPC call to the server is blocked till the call returns. In the current implementation of the GED, the RPC request service procedure is in the main thread of the server process. Therefore, even if two or more clients are making procedure calls at the same time they will be handled serially by the server, and a client may have to wait for service till the server finishes servicing the previous client request. This wait will be significant when the server is handling several clients and the events being delivered are large. In order to make the GED scalable it should be able to handle multiple client requests concurrently. Hence the first design goal is to have a multitasking

server, as shown in figure 5. Multitasking can be achieved either by multithreading or by forking child processes, as explained in [9]. In case of multithreading each RPC request will be serviced in a separate thread. When service procedures are being executed concurrently, two or more threads may be accessing the same data structures at a given time. To prevent race conditions, appropriate synchronization mechanisms must be provided for protection of data structures. However, locking of data structures must not be so coarse grained that it will effectively serialize their access. Hence synchronization mechanisms must be carefully chosen to a fine granularity of locking and to maximize concurrency. The GED server also is recoverable, which means that the server as well as clients can recover from crash in a way that is transparent to the other applications. This is achieved by using write-ahead log mechanism. In order to speed up the GED, a better format and algorithm is required for writing and reading the consumer log files. File access will also be quicker if log records are written in a binary format instead of ASCII as is currently being done.

3.2 Multithreading the Server

The GED server uses RPC and socket protocols in its communication interface. In order to listen for client requests when the server starts running, the server process first registers the program, procedures and version numbers with *registerrpc* command. The port mapper then advertises the availability of the RPC address so that interested clients can open a channel with the server. The server then goes to sleep while the *svc_run* call listens to the other end of a socket for a client request to come along.

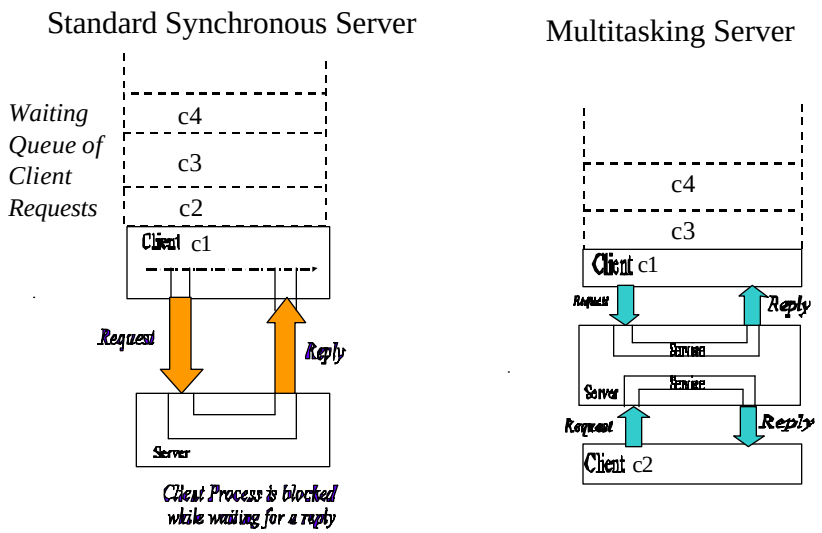


Figure 5: Synchronous RPC Server vs Multitasking Server

Based on the client request (argument), it executes one of the procedures mentioned in the *registerrpc* call and returns the reply (result) to the client. The *svc_run* routine is the heart of the server. Typically, it loops indefinitely, checking a set of socket descriptors. When it gets a service request, it switches to the associated procedure on examining the type of request. Once the procedure is executed, it loops back and waits for additional requests. Details of *svc_run* are in [8]. In order to make the server **scalable** it must execute each procedure in a separate process or thread, while the main thread (or process) listens for additional requests. Multi-tasking the server is useful mainly when there are multiple client requests that take widely different times to process. Even when the server is stuck in the middle of a request which is a long processing task , the server should be pre-empted by other brief or higher priority requests.

Multitasking can be achieved by either forking a process or allocating a thread to handle each request. In order to handle the request in a separate process, a child process can be forked during dispatching of the request. Another alternative would be to start a child process for each service procedure when the server is first started. In order to use multithreading instead of child processes, the *svc_run* routine must create a pool of threads where an idle thread will be allocated to handle the next request. The *rpc_control* utility function provided by the RPC library provides an option of starting the server in the AUTO_MT mode wherein the procedure dispatch routine in *svc_run* allocates a thread to handle each service request. Figure 6 shows the details of multithreading.

A thread is a single flow of control within a process. Threads share a single address space. Each thread shares the resources of the parent process. Although multitasking can also be achieved by creation of child processes, multiple threads of execution provide much higher performance as compared to full-blown forking [10]. First, since threads share global variables, memory sharing is not an issue with threads. Second, while context switching among threads, only pointer to the thread's stack and registers needs to be saved. For process context switches, all registers, stack, data, program counter as well as several runtime state parameters of process need to be saved. Hence context switching for threads is a lot cheaper than for processes. Third, when processes synchronize, they usually have to issue a system call, a relatively expensive operation that involves trapping into the kernel. But thread synchronization is usually handled by the runtime thread library, and is less expensive as it does not require a trap to the kernel. For the above reasons multithreading was seen to be the better of the two alternatives to achieve multitasking.

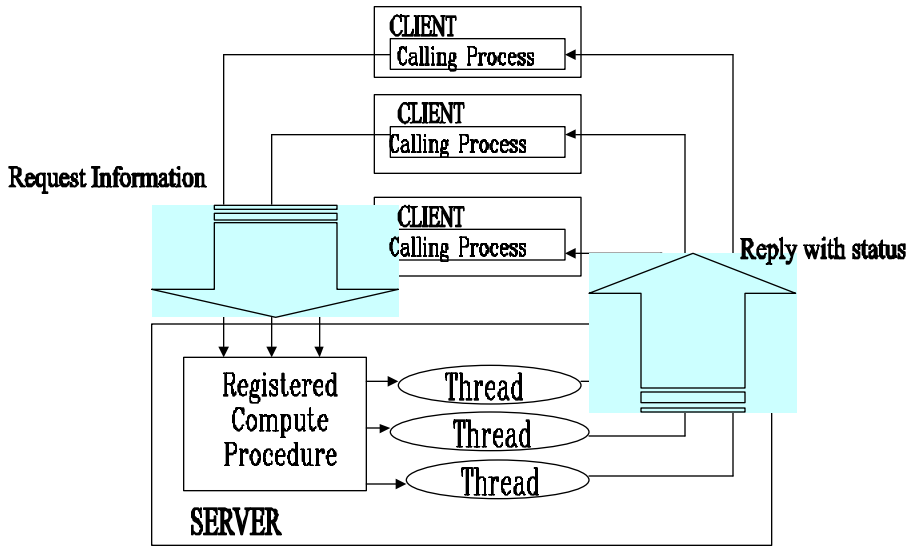


Figure 6: Details of Thread Handling

3.3 Synchronization Issues

The GED code is made up of several data structures that will be shared and hence may be concurrently accessed by threads. Following is the list of shared data structures:

1. Client address list (*client_addr_list):* list of client names and socket addresses.
2. Consumer event list (*event_para_list*): list of event consumer clients.
3. Global event graph (G_GED): graph of events with parameter lists to be propagated.
4. Producer list (*site_evnt_list*): list of producers.
5. Consumer list (*event_para_list*): list of consumers.
6. Event counter (*event_counter*): counter of total events received by the GED, used for assigning LSNs.
7. Event Log file (*consumer_name.log*): There is one event log file per consumer, that has one log record for each event occurrence to be received by the consumer.

8. Global event file (*GED_spec.log):* log of events in the GED.

9. Client Address file (*client_addr.log):* Client Ids and respective socket Ids.

Race Conditions. When the result of two or more threads performing an operation depends on unpredictable timing factors, there is race condition. Example of a race condition: Thread A is in the process of deleting a consumer node at position 7 from the *consumer_addr_list.* Thread B is traversing the *consumer_addr_list* to get the socket address of a consumer node at position 13 to which it wants to send a message. Thread B could be looking at node 7 when the list manipulation is occurring. Thread B will decide that node isn't the desired node and go to the next position in the list. However, since thread A has disconnected this node from the list the next position could be NULL. The result of what thread B reads will hence depend on the timing factor and has been compromised by the race condition. Hence the access of the *consumer_addr_list* and several such shared data structures must be guarded for mutual exclusion. This can be attained using synchronization mechanisms or locks. There are several types of locks and the right choice must be made.

3.3.1 Types of Locks

Mutex lock is a synchronization primitive that allows multiple threads to synchronize access to shared data by providing mutual exclusion. The mutex lock has only 2 states: locked and unlocked. Once a thread has acquired the mutex lock on a data structure other threads attempting to lock the structure will be blocked until it is unlocked. Since mutex allows only one thread to access any data at a given time, it is the most restrictive type of access control. For example, when a mutex is used to synchronize access to a list, the mutex will control the entire list. While the list is being accessed by

one thread it is unavailable to all other threads. If most accesses are reads and writes of the existing nodes as opposed to insertions and removes, then a more efficient approach will be to allow nodes to be individually locked.

Read-write lock is another synchronization primitive that was designed specifically for situations where shared data is read often by multiple threads/ tasks and rarely written. A read-write lock is similar to a mutex lock except that it allows multiple threads to concurrently acquire the read lock whereas only one writer at a time may acquire a write lock. In the current scenario the *Insert* or *delete* operation on a list will require acquiring the read-write lock in the *write_lock* mode, while the seek (search) of a node will require acquiring the lock in the *read_lock* mode. By using the read-write locks we can have parallel search operations in the GED. The only drawback of using read-write locks is that locking operations take more time than the locking operations on mutexes. Hence locking strategy must be chosen carefully. Read-write locks are justified for the *event_paralist* and G_GED data structures in the GED where inserts to the data structures happen only once at the beginning when clients connect with the server; thereafter all other operations are search operations on the list to find a particular node. *Read_lock* mode can be used to allow threads to search the list in parallel.

Semaphore is a synchronization primitive that has a value associated with it, which is the number of shared resources regulated by the semaphore. Whenever a thread acquires a semaphore, the semaphore count is decreased by 1. Whenever a thread releases a semaphore, its count is increased by 1. Any thread wanting to acquire the semaphore must wait till its count is greater than 0. Traditionally, semaphore operations have been known as P and V operations. P operation is equivalent to acquiring the

semaphore (*sema_wait).* V operation (*sema_post)* is the same as releasing the semaphore. Semaphores are used primarily when there is more than one shared resource that needs to be regulated.

For synchronization of data structures in the GED, mutex locks or semaphores can be used when the operations involved are primarily inserts and deletes that require exclusive access. For data structures such as the *event_para_list*, where a majority of the operations are search operations on the list and updates on individual nodes, read-write locks can be used for locking the list and semaphore or mutex locks can be used for locking individual nodes. Details of the locking algorithm are explained in the implementation section. The table below shows the choice of locks made for locking the various data structures with reasons for the choice.

Table 1: Locks used for Lists and Log Files

DATA STRUCTURE and CHARACTERISTICS	LOCK USED with RATIONALE
1. *Client_addr_list:* list of client socket addresses. Nodes are inserted into the list when a client joins and are deleted when a client *unregisters* with the GED. Whenever the GED has to send a message over the socket to a client, this list is scanned	Mutex locks are used as operations are primarily inserts or deletes which happen when a client joins or leaves. These operations need an exclusive lock mode which is provided by the mutex lock. Since each node only has 2 fields, scan of list is a fast operation. Using mutex locks is preferred to read-write locks, also because operations on read-write locks have a high overhead
2. *Event_para_list:* List of consumer event nodes. Nodes hold the events that are received by the GED and pulled by consumer clients. There is one node (*event_noti_node*) per consumer	Read Write locks are used for locking the list and semaphores are used for locking individual nodes as operations on the list are primarily search of the list to find an individual node, followed by updates on that node's contents. Shared mode (read lock) can be used while scanning the list to allow parallel scans and exclusive mode (write lock) is needed when nodes are inserted or deleted from the list. Semaphores are used for locking individual

	nodes as updates must be done exclusively.
3. *Consum_list*: list of consumers that connect with GED	Mutex locks are used as operations are primarily inserts or deletes when a consumer joins or leaves.
4. *Event_ptr_l*: list of event nodes.	Mutex locks are used as operations are primarily inserts or deletes. Since scans are very fast, mutex operations which are faster are preferred to operations on read-write locks.
5. *G_GED:* global event graph. This graph is traversed and parameter lists are propagated when event occurrences are sent to the GED.	Read-write lock for locking G_GED list. Write lock provides exclusive access to graph while inserting or deleting a node. When accessing list in shared (read) mode, lock hash table is used for managing access to individual nodes. Lock hash table minimizes number of semaphores needed to lock nodes of the G_GED. Thread suspend and continue calls are used to prevent more than one thread from accessing any node at a time. Lock table minimizes overhead of managing several locks.
6. Site_event_list: list of producers. There is one node per producer that holds the list of events that the producer must send to the GED	Mutex locks are used as scan operations are fast, and operations on mutex locks have a lower overhead than operations on read-write locks. The list is accessed only when a producer connects with the GED or recovers from crash, so operations on the list are also less.
7. Global log file (ged_spec.log). This file is used for construction of the G_GED graph when GED recovers from crash.	Mutex lock as file read or write must be mutually exclusive.
8. Event_counter: count of events received by GED.	Mutex locks as counter is constantly updated when new events arrive at GED
9. Consumer event log file	Each consumer event log file is locked separately by using Semaphore locks. Like mutexes, semaphores provide exclusive lock mode.

3.4 Improving I/O for Logging and Recovery

To provide recovery three log files are maintained by the GED, as explained in [11]. The consumer log file is given by *consumer_name.log* and is the file to which event occurrences sent to the GED are written before they are placed in the main memory buffer. In case of the crash of a consumer, events sent to the GED continue to be written into the consumer's log file. When the consumer client recovers after a crash, unconsumed events are read from the log file into the main memory buffers from where they are read by the consumer. Each log record has is identified by a unique LSN which is its log sequence number. At present the log file maintains two pieces of information in its header: *BLSN* is the log sequence number of the last event in the buffer. *DLSN* is the log sequence number of the last event that has been pulled by a consumer. The current algorithm for reading from the log file starts with reading the *BLSN* (during recovery it is DLSN). Thereafter, every record in the file is read until the record is reached whose LSN is greater than *BLSN*. After reading this record into the buffer, the *BLSN* value is updated. Thus, in order to read any record all the previous records have to be read first, which gets time consuming especially when the file is large. To speed up log file reads two new header fields are introduced. The file offset for the *BLSN* and *DLSN* can also be maintained as a part of the header. In this way read of a record will involve a read of its *BLSN* and *blsn_seek_offset*, followed by a *seek* into the file to the desired record. Records can also be read and written using the binary *fread* and *fwrite* which is faster than writing in ascii where *fprintf* and *fscanf* are used. Details of algorithm for speedup are explained in the implementation section.

CHAPTER 4
IMPLEMENTATION OF MULTITHREADED GED

As discussed in previous sections the RPC service request routines are threaded so that the GED can handle different procedure requests concurrently. In addition, some other procedures in the GED process are also threaded to give better performance, especially when the GED is running on a multiprocessor machine. This chapter discusses the details of the threading implementation. In the previous chapter, synchronization issues for the various GED data structures had been discussed. This chapter gives a detailed description of locking mechanisms used to synchronize access to the G_GED (global event graph) and the consumer event list data structures. In order to enhance performance of GED, changes were also made in mechanisms of buffer management and logging, which have been described in detail. Finally, the implementation of the *shut_server,* which was designed to gracefully shut down the GED is discussed.

4.1 Threading of RPC Procedures

The RPC library provides ***rpc_control*** function for multithreading purposes. It provides a framework for the server to create threads for the RPC function calls. When this *rpc_control* function is called with “Automatic” as the argument, a thread pool, whose default size is 16, is generated by the server dispatch routine *svc_run.*. When a request arrives, a thread from the pool will be activated to handle that request. Subsequent requests will be queued up if all the threads from the thread pool are busy.

Following is a list of the RPC procedures:

1. ***global_reg*** : This thread is created for the RPC call where a client sends the name list of events (that it needs from other clients) to the GED, leading to the update of the Global event graph, G_GED.
2. ***namelist_update*** thread is created for the RPC call where the producer client updates its name list of events in order to know which events it should send to the GED.
3. ***global_notify*** thread is created for the RPC call where the client sends an event to the GED and the consumer event list (*event_para_lis)t* is updated.
4. ***recieve_notify*** thread is created for the RPC call where the client pulls events from the GED by reading them from the *event_para_list*.
5. ***Unregister*** thread is created for the RPC call when a client unregisters with the GED.

4.2 Additional Threads in the GED

Additional threads were introduced to give a higher performance for the GED, especially when the GED is running on a multiprocessor machine. These threads are listed below:

1. *get_client_addr* is the handshake thread. The GED must continuously listen on a socket for incoming client connection requests. The handshake thread was introduced to listen for client requests on the socket. When the GED receives a message on the socket, it is parsed to determine the type of request. Requests are of the *init, resume, unregister* or *shut* types. In i*nit* mode client connects for the first time, if a log file already exists for the client, it is unlinked. In *resume* mode the client reconnects after a crash. Its socket ID in the client address list is updated. In *unregister* mode client unregisters and will not reconnect in resume mode. The *shut* message is sent to gracefully shut down the GED.

2. *sendback_event_name_fun* thread updates the *namelist* for each producer site after the consumer notifies events of interest to GED. It updates the list of events for which the producer must notify to GED.

4.3 Locking of Global Event Graph (G_GED)

Global events are detected on the server using an event graph. An event tree is created for each composite event and the trees are merged to form an event graph for detecting a set of composite events. Leaf nodes represent global primitive events and the non-leaf nodes represent global composite events. For each node in the event graph there is an event subscriber linked list containing all the composite events that use this event as its constituent event. Each node, which is an operator, has a pointer to each of its child nodes for which it is an event subscriber. The parameter list *(L_OF_L_LIST*) is the list of event parameters. It contains the signature and values of arguments of the method that raised that event. When a global primitive event is detected at the leaf node, its parameter list is propagated to its subscribers, which are the parent nodes. By propagation we mean that the *L_OF_L_LIST* will be examined and copied or merged with another *L_OF_L_LIST* at the parent node, depending upon the type of composite event. Event occurrences flow upward to the root. If the composite event occurs for the current event notification, the composite event is detected. This detected composite event then further propagates upwards as a constituent of the composite event to be detected at its parent. Traversal of the G_GED was discussed in chapter 2 in more detail. In a multithreaded server, several threads of execution share the G_GED graph, and access to the graph has to be synchronized. Using a mutex lock for the G_GED locking will give only two states (locked and unlocked) of access for the entire graph so that only one thread can be

accessing it at any time. To give finer granularity, more than one thread should be able to access independent nodes of the graph concurrently, as long as they are not updating the same nodes.

One way to achieve a finer granularity would be to have a read-write lock on the global event graph and a semaphore lock on each node of the tree. However, when a large number of clients connect with the GED, the number of nodes in the tree will grow as each node represents a primitive or composite event or an intermediate node in the composite event tree. Allocating and maintaining locks for each and every node of the tree is cumbersome and will require too many locks. A better option is to maintain a hash table of the nodes of the tree that are currently being accessed. Each node of the tree will hash to a bucket of the hash table. The bucket will maintain a list whose elements represent the Ids of nodes of the G_GED currently being accessed. Thread IDs of threads waiting for a particular node will also be saved in a queue for each element in the list. In order to traverse the list of node IDs the bucket needs to be locked. This means that the maximum number on semaphore locks required for synchronizing access to the G_GED is equal to the number of buckets. In this way number of locks to be maintained is minimized and at the same time a fine granularity of locking is achieved for locking the G_GED.

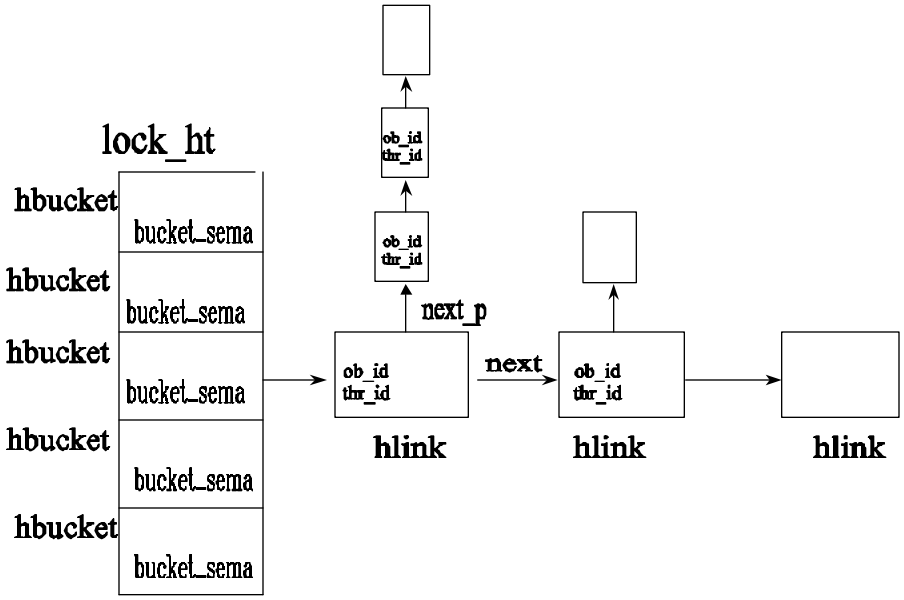


Figure 7: Lock Hash Table Data Structure

The classes defined for the hash table are *lock_ht* , *h_bucket* and *h_link*. *Lock_ht* is the lock hash table class. There is a single instance of this class for the GED. *h*_bucket is the class for each bucket of the hash table. Each bucket is guarded by a semaphore *bucket_sema,* and each bucket contains a chain *of h_link.* The *h_link* contains *obj_id, thr_id* , *next* and *nextp*. *Obj_id* is the unique ID (address of the node is used for hashing) of the G_GED node being accessed by a thread whose thread id is *thr_id.* *next* is a pointer to the next *h_link* in the bucket chain. *nextp* is a pointer to an *h_link* which contains the *thr_id* of a thread that is suspended and waiting to access the same G_GED node. Figure 7 gives the data structure of the lock hash table.

When an event occurrence is sent to the G_GED, the G_GED is traversed and parameter lists of one or more individual nodes will be updated. For traversal, read-write locks are used to give shared access to the G_GED. Lock hash table is used to access

individual nodes when their parameter lists are being propagated. Each node of the G_GED must have a unique object ID for hashing purposes. Since the address of the node is unique, it is used as the object ID. The sequence of operations needed for locking are as follows: First the node object is hashed to find its bucket in the hash table. A semaphore lock (*bucket_sema)* is then acquired on the bucket so that no two threads may be accessing its chain of *h_link* at the same time. The bucket chain is then searched for the object ID of the G_GED node. If the object ID is not found, it means that no other thread is accessing this node. Then an *h_link* containing that node's object ID and thread ID are added to the bucket's *h_link* chain, the bucket semaphore is released, and the current thread is granted access to the G_GED node. The thread can now copy or modify the node's parameter list. On the other hand, if the object ID is found in the chain, it means that another thread is operating on the node at the same time. The current thread's thread ID is added to the list of waiting threads for that G_GED node. *bucket_sema* is released once the object ID is located, so that other threads can traverse the *h_link* chain for accessing nodes of the G_GED. The current thread is suspended and will be continued only when the desired G_GED node becomes available to it. After a thread finishes accessing the G_GED node, it removes its thread ID from the *h_link* chain. The next thread in the queue of suspended threads is released by a "*thr_continue*" and it can now access that G_GED node. Figure 8 gives the locking algorithm.

LOCKING NODE OF G_GED

- bucket_id = hashing(object_id)
- P(bucket_sema(bucket_id))
- search h_bucket for object_id
- Found object_id:
  - insert into vertical chain(queue of waiting threads)
  - V(bucket_sema(bucket_id))
  - suspend(current_thread_id)
- Else
  - insert into bucket chain
  - V(bucket_sema(bucket_id))

RELEASING NODE OF G_GED

- bucket_id = hashing(object_id)
- P(bucket_sema(bucket_id))
- search bucket chain for object_id
- if next_p != null
  - V(bucket_sema(bucket_id))
  - continue the next thread in waiting thread queue
- Else
  - delete hlink from chain
  - V(bucket_sema(bucket_id))

Figure 8: Locking Algorithm for G_GED nodes

4.4 Locking of Consumer Event List

The *event_para_list* is the list of consumer nodes (*event_noti_node*) that carry event instances that are pulled by consumers. Figure 9 gives the data structure of this list. When producers send event instances to the GED, they are inserted into the list and they are pulled from the list by the event consumers. To have a fine granularity of locking and to provide maximum concurrency in access of the consumer event list (*event_para_list*), read write locks are used. As operations on the list are mostly read operations, having read-write locks for *event_para_list* will give maximum shared access to the list, and at the same time semaphore locks can be used for exclusive locking of the individual nodes when they are being updated. Sequence of operations for locking are as follows:

1. First time an event arrives for a consumer, create a new *event_noti_node* for that consumer. Then *write_lock* the *event_para_list* and insert the *event_noti_node* into the *event_para_list.*
2. Every time events are read from the *event_noti_node* (in *recieve_notify* call): Read lock *event_para_list*, seek to consumer's *event_noti_node* and release read lock. Then lock *event_noti_node* and copy its *para_list* for sending. Then initialize the *para_list* by *re_init()*. Finally release semaphore lock on node.
3. When a client crashes: Read lock *event_para_list*, seek to the consumer's *event_noti_node* and release the read lock. Lock *event_noti_node*. Then initialize its *para_list* by *re_init()*. Finally release semaphore lock on node.
4. Client unregistered: write lock *event_para_list*. Delete the *event_noti_node* for the consumer. Then write-unlock *event_para_list.*
5. Client receives event: The node is no longer deleted on a *receive_notify* call. Only its *para_list* is reinitialized. At this time a read lock is obtained to seek (search) the node in the *event_para_list* and then a semaphore lock is attained on the individual node while modifying its *para_list* via the member function *re_init()*. Finally semaphore lock on the node is released.
6. Client unregisters: write lock *event_para_list*. Delete the *event_noti_node* for the consumer. Then write-unlock *event_para_list.*

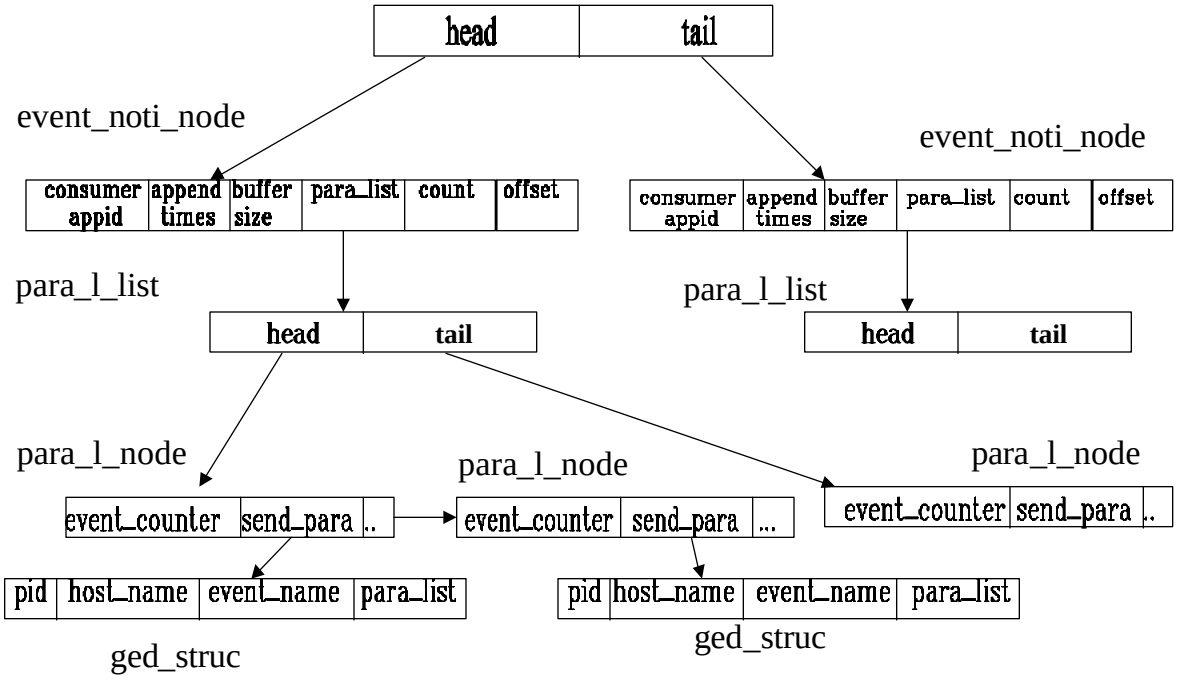


Figure 9: *event_para_list* Data Structure

4.5 Performance Improvements in Buffer Management

Each consumer node (*event_noti_node*) of the *event_para_list* contains a *para_list* which is the list of event parameters. In the earlier implementation, every time the *para_list* was read from the consumer node through the *receive_notify* RPC call, the entire node was deleted from the list. Each time a new event was to be added, a new node had to be constructed and inserted into the list. This had an overhead of creating or destroying the entire consumer node each time it was accessed. This also added an overhead of locking the entire list for each insertion or deletion of the *event_noti_node*.

To improve concurrency, that is to reduce the number of times the list is write locked, the consumer node is inserted into the list *only the first time* a new event arrives for the consumer. The node is no longer deleted when an event is read (*recieve_notify*) from the node. Only the node's *para_list* (which is the list of events within the node) is reinitialized by using a new function, *re_init()*. On a client site crash, the *para_list* is again reinitialized and the appropriate parameters (such as crash_flag) are set in the consumer node to signal a crash. Buffers held by this consumer are then released. However, the event_noti_node for this consumer is not deleted. This node is deleted only once when the consumer exits the scene with an *unregister()*.

4.6 Performance Improvement in Logging

For the purpose of persistence and recovery the GED uses a write ahead log. Each consumer is assigned a unique log file given by *consumer_name*.*log*.

Event received by the GED for a consumer is first written into the consumer's log file and then inserted into the *event_para_list*. The log file serves two purposes. When the producer sends events to the GED at a rate much higher than the consumer's rate of event consumption, the extra events are read from the log file when the consumer becomes free. Each record in the log has a unique log sequence number (LSN) and the event parameters. Every event sent to the GED is first written into the log file and then inserted into the consumer's buffer, which is a node of the *event_para_list*. *BLSN* (buffer LSN) is the LSN of the last event in the buffer. If records need to be read from the file into the buffer, then the file needs to be searched for the record that has the lowest LSN greater than the *BLSN*. *DLSN* (delete LSN) is the LSN of the last record that was pulled by the consumer from the buffer. This number helps in recovering from GED

crash. Every time the consumer pulls an event, the *DLSN* for that consumer is updated in its event log file. If the GED crashes, then all information of events in the buffer is lost. On recovery, it must read the *DLSN* in order to know which was the last event received by that consumer. BLSN is then made equal to the *DLSN* and events are read from the log file into the buffer as before. Similarly, when a consumer crashes, its buffers are freed after a certain fixed amount of events (*append_times*) are written to the log file and not pulled from the buffer. To recover from the crash, its *DLSN* is read and its *BLSN* is made equal to its *DLSN*, after which events with LSNs higher than *DLSN* are read into the buffers. Writing records into log files takes a considerable I/O time. In order to speedup reads and writes all data is written in the binary form using functions *fwrite* and *fread* instead of writing it in ASCII format using *fprintf* and *fscanf*. To speed up access of a record given its LSN, two new fields were introduced in the header of the event log file. Two new fields were introduced in the header: *BLSN seek offset* and *DLSN seek offset* with which it becomes possible to seek to the exact position of the record in the log file, given the *BLSN* and *DLSN* respectively. The header now has the following fields:

dlsn: log sequence number of the last event received by the consumer

blsn: log sequence number of the last event in the buffer.

blsn_seek_offset: offset position (from the start of file) where the BLSN record ends.

dlsn_seek_offset: offset position where the DLSN record ends.

The functions used for manipulating log files are *event_file_insert, event_file_to_list* and *event_file_delete.* These functions and their modifications are as follows:

1. ***event_file_insert:*** In this function an event is written into the log file. The open file descriptor and *L_OF_L_LIST* are passed to this function. Event information is

appended to the end of the file. The entire *L_OF_L_LIST* is written into the file. Three structures: *copy1, copy2* and *copy3* were introduced for writing the list. Since the class *L_OF_L_LIST* contains several lists of unknown lengths, each time the length of a list is written first, followed by all the nodes of the list, which are of fixed length. While reading the list from the file, the length will be read first and then the nodes. Seek offset position from the top of the file where the inserted record ends is obtained using the function *ftell*. The function returns this offset position. This position information becomes a part of the *event_noti_node* if event is inserted into the *event_para_list*.

2. ***event_file_to_list***: This function is used for reading events from the log file into the buffer. When a producer sends events at a speed faster than the speed at which a consumer receives them, the consumer buffer gets full. Write ahead log makes sure that the events are persisted, whether the buffer is full or not. At a later point, when the consumer removes events from the buffer, events that could not be stored in the buffer are read by the GED from the log file into the buffer and the *blsn* and *blsn_offset* value in the file is updated to reflect the LSN of the last event read into the buffer. When a consumer crashes, it no longer receives its events. After detecting the crash the GED frees its buffers and thereafter, events for this consumer are only written to the log file. They are read from the log file once the consumer recovers. When the GED fails, it may have some events in the consumer buffers, which were not yet consumed by the consumers. On GED recovery, the *DLSN* is used to read unconsummed events from each consumer's log file. For each consumer, the *Event_file_to_list* function is passed the open file descriptor and the offset position in

the consumer's log file from where the event is to be read. Data is read into structures. After reading an event into the buffer the *BLSN* and *blsn_seek_offset* are updated to reflect the last event read.

3. ***Event_file_delete*:** When the consumer removes an event from the buffer, the log sequence number and seek offset are read from the *event_noti_node*. These are used to update the *DSLN* and *dlsn_seek_offset*. On the crash of a client or GED failure, *BLSN* and *blsn_seek_offset* are made equal to *DLSN* and *dlsn_seek_offset* respectively.

4.7 Design of Shut Server

A *shut_server* has been implemented to shut down the GED. Host and port on which the GED is running are passed as arguments to the *shut_server*. The *shut_server* establishes a connection with the GED process on a specified port and sends a "*shut*" message over the socket to the GED that is listening for client requests in the *get_client_addr* thread. Command used to shut the server is:

shutserver –m[ged machine name] –p[port name]

On receiving the message, the GED parses the request to determine the request type. The GED then goes through a clean up procedure before exiting. In the clean up: 1) GED_spec.log, client_addr.log and each of the consumer event log files are deleted. 2) *.recover* file (used to detect GED recovery from crash) is deleted. 3) memory is freed for the buffer manager and data structures. In this way the GED gracefully exits by releasing all its resources.

CHAPTER 5
PERFORMANCE EVALUATION OF THE ENHANCED GED

This Section outlines the tests done to measure performance enhancements in the multithreaded GED server. Earlier the GED could handle only one RPC request at a time. When multiple clients connected to the GED and sent concurrent requests, the requests were effectively serialized. With the multithreaded implementation the GED can handle several requests concurrently in separate threads. This is expected to effectively reduce the response time, which is the difference between the request made by a client to the reply received by it from the server.

Profiling a program is a meaningful approach to identifying its performance bottlenecks. To track the time that the server spends using the CPU or waiting for locks and I/O completion, the profiling tool *prof* available on UNIX was used. Throughout the testing of the GED implementation, the *prof* tool was used to look at the times spent by the various operations in GED and to make adjustments in the program. For example, it was seen using the *prof* tool that in the server process a significant time was being spent in waiting for the lock for the event log files. To reduce this wait the lock allocation was changed from one mutex lock for all the event log files to one lock per event log file for each consumer. This reduced the response time at the consumer as each consumer log file could be locked separately and locking was independent of other event log files so waits were reduced.

To set up test programs for experiments we create a specialized producer client program that can send the server a stream of events and measure its response. A consumer client program is also created that subscribes for the same set of events with the GED and has rules defined on them. The test client measures the total time it takes to complete a large number of event notifications. For the producers these operations are the send of event occurrence to the server; for the consumer the operations are primarily the receive (pull) of events from the server. For the experiment, multiple client processes issue requests to the server across multiple connections.

To evaluate multithreading, we run the GED server in two different modes – a serial server (one that does not use threads at all – runs in single threaded mode) and a multithreaded server. The GED server threading mode can be specified in the configuration file before starting the server. When running in the multithreaded mode, the number of threads in the thread pool to handle the requests is also varied. The server is first run in a single threaded mode and number of threads are then increased to 16, 32, 64 and 100.

The *persist* or *nopersist* mode is another factor that affects the performance on the GED server. In the persist mode the GED server uses a write ahead log to provide client as well as GED recovery. It writes every event received from the producer into a log file using the function *event_file_insert*. Records may be read from the file using the function *event_file_to_list*. The main difference in the two modes is that in the persist mode the request service procedures are I/O intensive. Another major difference is that in the *nopersist* mode there is no buffer manager to hold the excess events. Every event received from a producer has to be sent to the consumer of the event before the next

event for that consumer is accepted. In other words in the *nopersist* mode the producer cannot send at a rate higher than the receiving rate of the consumer, and procedure calls made by producers or consumers are effectively serialized. In the persist mode, the rate of send by the producer is independent of the rate of receive.

Performance of the GED depends on the number of clients and contention. The type of events that the consumers subscribe to affects contention. For example, if we have a set of consumers C1, C2, C3, C4 that subscribe to a set producers P1, P2, P3, P4 respectively, where each producer generates a different event, then all the 4 consumers can be consuming events in parallel with no sharing as each will be accessing a different event buffer of the consumer event list. Contention will be less in this scenario. On the other hand if all the four consumers C1, C2, C3, C4 subscribe to the same event from a single producer P1, contention is high. The number of clients being serviced by the GED also affects contention. As the number of clients connecting with the GED increases, the contention increases.

The speed of event generation by the producer clients also affects the time difference observed between single and multi threading modes. If the producer generates events with a significant delay between each event (1-2 seconds) then this interval is much more than the time required for processing the event. Therefore the response time will almost be the same, whether the GED is run in the single or multithreaded mode. On the other hand when events are generated with no delay between them response times will vary for the single and multithreaded modes.

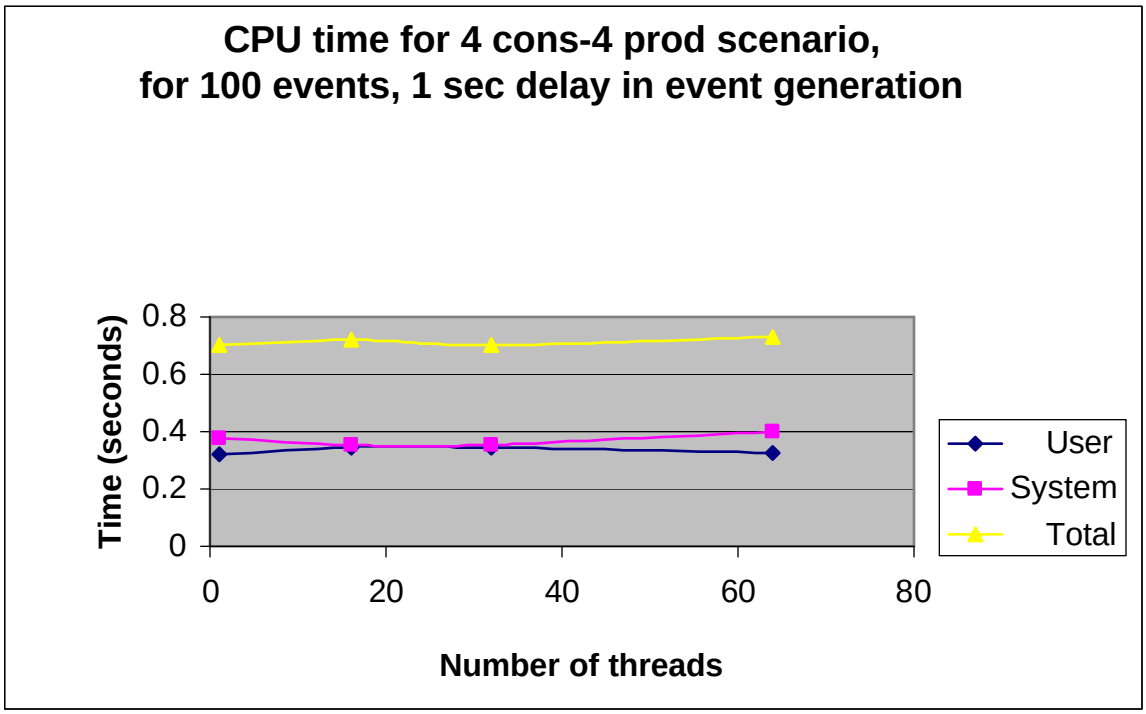
5.1 Experimental Setup

In order to have multiple clients, tests were run with 4 consumer and 4 producer clients that connected with the GED. Tests were performed on an 8-CPU multiprocessor machine. All clients were started together and connected with the GED for each set of readings. To get an accurate measure of readings, 10 readings were taken for each test case scenario by running it 10 times and the average was calculated. In order to have a large number of events to the GED, each of the 4 producer clients was made to generate a 100 events. Since the buffer size of the event buffer was kept low to 5 buffers, a 100 events by each client was a sufficient value to test performance of the buffer manager under heavy load. Different sets of readings were obtained when running in the persist mode as the number of threads was increased from 1 to a 100. A timer was started at each client when it received the first event and stopped when it finished receiving the last event. The time value obtained was divided by number of events to find the response time of a single RPC call.

The server's response to a client's request involved different amounts of I/O and more or less CPU intensive tasks. The different times measured at the client were:

Response time: Time between the request and reply at the client. This was used to measure the performance gain. *System time* is time spent in CPU intensive system tasks. *User time* is time spent in I/O. *CPU time* was the total of *system* and *user* time. The timing functions used were *gettimeofday* to measure the response time, and *getrusage* to measure the CPU times. The readings obtained for the different scenarios are shown below.

Threads	User	System	Total	Response time
1	0.3214	0.3786	0.7	98.6587
16	0.346	0.354	0.72	98.52
32	0.345	0.355	0.7	98.081
64	0.327	0.4	0.73	98.45
100	0.338	0.362	0.72	98.52



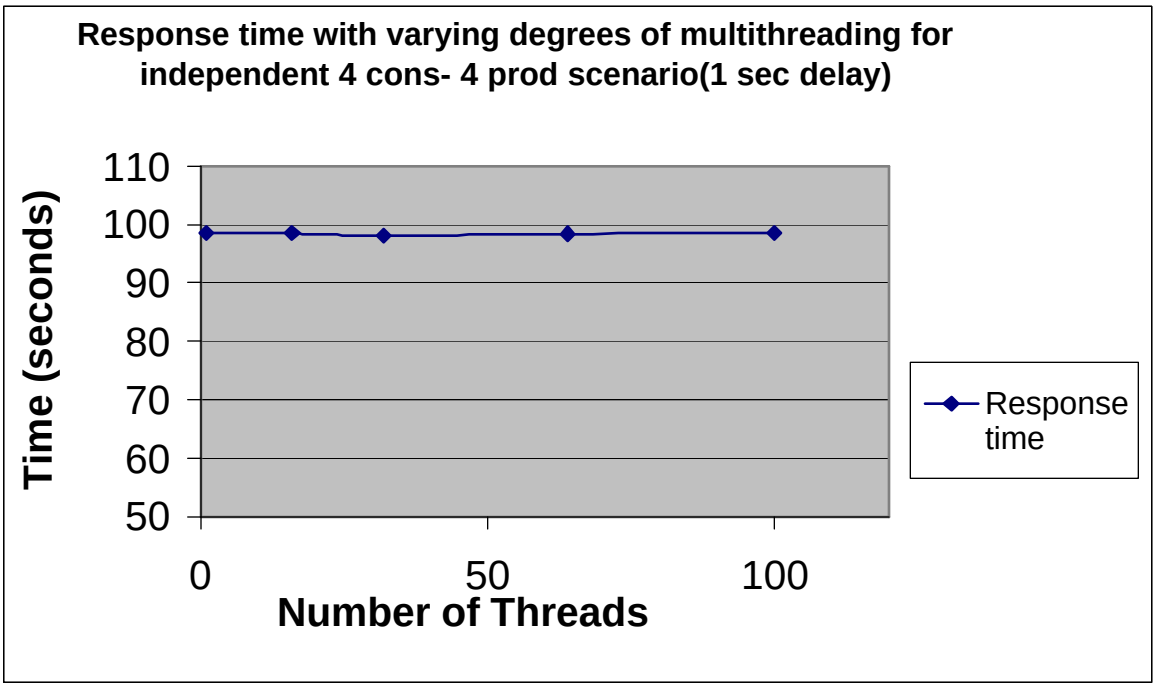


Figure 8: Time Measurements for 4 prod- 4-cons (1 sec delay in event generation)

For 4 producers and 4 consumers subscribing to independent events, the results are shown for 1 second delay between event generation in figure 11. These readings are taken in persist mode. The total CPU time remains almost the same even if the number of threads is changed because the total amount of CPU power needed for processing is the same. The GED has a certain capacity for processing events. It takes a certain maximum amount of time for the processing. If the delay between the events sent to the GED is more than this maximum then even if the GED is run in multithreaded or single threaded mode the processing will always be done within this time. Since the delay of 1 seconds is more than the time required for processing an event, the response time doesn't change much by changing the number of threads for this test case.

In the 4 consumer- 4 producer case (Figure 12), when there is no delay in event generation with consumers C1, C2, C3, C4 subscribing to events P1, P2, P3, P4 the response time shows a drop at the first reading for the multithreaded case (16 threads). This is because the when 16 threads are spawned to handle the request, processing can be done in parallel. As the number of threads is increased, contention and wait time for locks increases and is an overhead, thus response time again increases for 32 threads. Increasing number of threads beyond a certain point does not improve performance further as the CPU power available for the process cannot increase.

For the 4 consumer-1 producer case, the response time drops initially upto 32 threads and then becomes steady as increasing number of threads will not improve performance further; As opposed to the 4consumer – 4producer case, where there can be 8 or more threads trying to update the buffers at anytime, in the 1 producer- 4 consumer case there is only 1 producer that will be adding events to the buffer. The 4 consumers

can be consuming events in parallel. As contention is less in this case, response time becomes steady even as the number of threads is increased. Figure 13 shows the time measur**Time**ements for this scenario.

Threads	Response time
1	17.9912
16	13.2475
32	16.552
64	15.4125
100	16.09

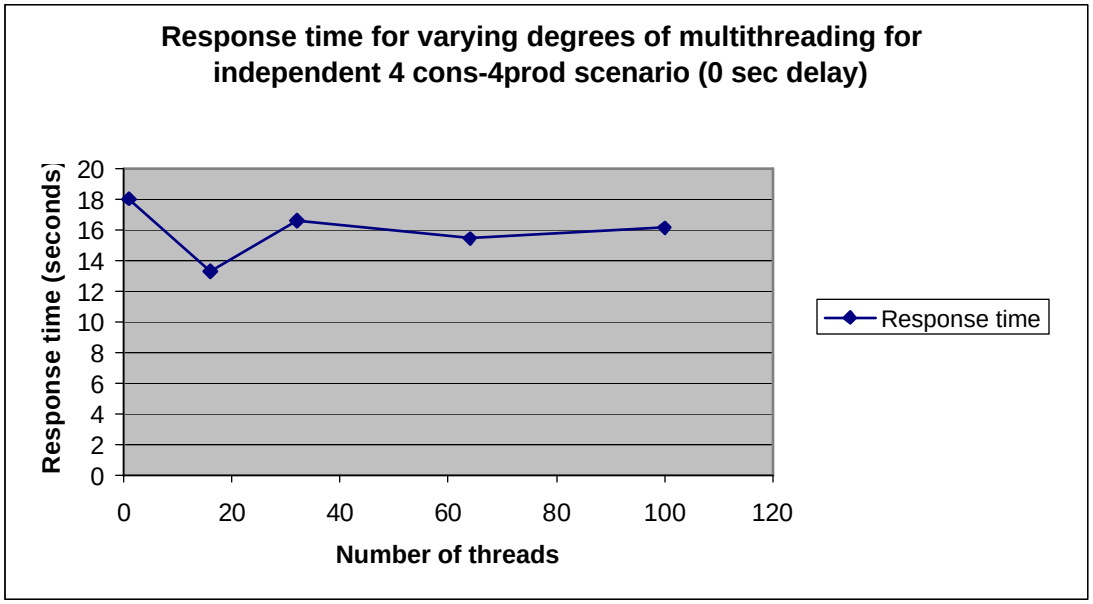


Figure 8: Time Measurement for 4prod-4cons (0 sec delay in event generation)

Threads	Response time
1	20.1
16	18.94
32	14.93
64	13.45
100	13.56

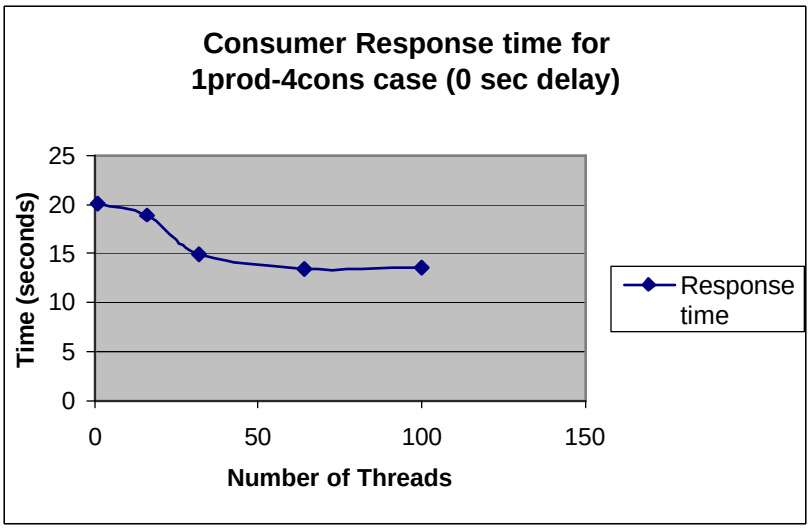


Figure 9: Time Measurement for 1prod-4cons Scenario

As shown in figure 14, producer clients benefit the most with the threaded GED. This is seen by measuring the producer's response time (time between the point when the producer starts producing the first event to the time at which it returns after finishing sending the last event to the GED). GED is running in persist mode on eclipse which has 6 processors and the producer is generating events for 4 consumers. The producer generated 100 events in a second The response time drops heavily when there is a switch from the single to multithreaded mode (16 threads). This is because producer generates all the events at once; of the 16, threads created, upto 3 may be used by the consumer in making an RPC call to pull the event. The rest of the 13 theads are all available to the

producer for doing the event notification in parallel.. Time measurd for single threaded case was 21.5 seconds. If this time is divided into 13 threads, then a fall to 1.9.seconds is expected. The response time for multithreaded case remains around 2 seconds as expected and gets steady as the number of threads are increased because the total processing power allocated to the server process is constant.

Threads	Response time
1	21.5
16	2.705
32	2.44
64	2.46
100	2.35

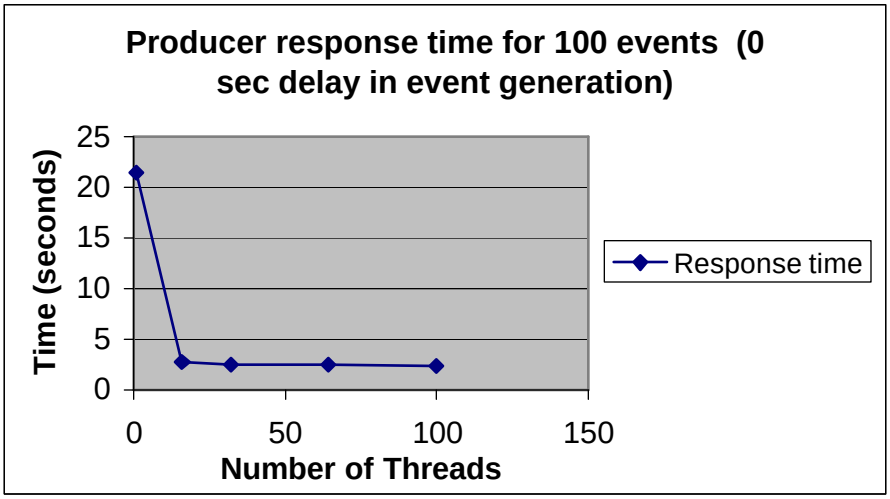


Figure 10: Time Measurement for prod Application

5.2 Summary

The results obtained in the above test scenarios (Figure 11-14) show that improvements measured in response time on switching from single to multithreaded case, are most significant in the cases where consumers subscribe to independent events. The benefits measured in terms of response time are best seen by a producer of events who

can generate all its events and continue to do other useful work without being affected by the speed at which events are received by the consumer. For consumers a drop in response time is observed in moving from single to multithreading modes. When events are generated with 0 delay by producers, rate of generation is higher than speed of consumption and events may have to be read from log files. In this case the number of also I/Os also increases the response time measured at the consumer.

CHAPTER 6
DESIGN ISSUES FOR RULE SUPPORT

In the current implementation, the GED is merely an event mediator, passing global events across applications. This means it can be used readily for purposes such as stock applications where the price decrease in IBM stock causes a rule to be triggered in the stock application, and this event is propagated without any delay by GED to a "stockbroker" application. "Stockbroker" application fires a rule to purchase the stock immediately at that price.

If the GED was to be used in situations such as data warehousing, where data from various heterogeneous sources is collected, filtered, and sent to a warehouse for integration, capabilities of the GED have to be enhanced. GED has to be augmented to do certain useful computation on the received global events or delay the propagation of events to the consumer applications. Thus a design goal is to add the capability of rule support on global events at the GED. Supporting rules will involve 1) design of a rule editor for rule specification, 2) back end rule server for rule persistence and management, and 3) a dynamic loader for loading rules into the GED address space at runtime.

6.1 Extensions to the Graphical User Interface

To meet the requirements of multi-platform usage and portability, Java has been chosen as the implementation language for the interface module of the editor. At present a rule editor interface exists for specifying rules on events local to applications. This interface is extended for specification of rules on global events. The opening window will

now allow the user to select between a choice of global and local rules. Based on whether Global or local rules were selected, appropriate chain of windows will follow to give the user convenience in editing rules.

6.2 Architecture

Server architectures can be categorized as 2-tier or 3-tier based on how the client/server application can be split into functional units. The typical functional units are the user interface, business logic and the shared data. In two-tier client/server systems, the application logic is either buried inside the user interface on the client or within the database on the server. A three-tier architecture augments traditional client/server computing by introducing a middle tier component [12]. There are three component layers: The front-end component which is responsible for providing portable presentation logic; the middle-tier component, which has the application logic and allows users to share and control business logic by isolating it from the actual application; and the back-end component, which provides access to dedicated services. Three-tired client server architecture is used for the rule editor. The main reason why a three tired architecture is chosen is that the processing of the requests coming from the interface often gets complex and is best performed on a dedicated application server rather than by the database server. Three tier applications are more scalable because they minimize the load on the server. Extending a three-tier server is also less complicated than extending a two-tier server. The user communicates with rule editor at the topmost layer. At the middle layer there is a rule server, similar to an application server which will do the processing or the rules, persist them into files and service requests from the editor (client to rule server) for retrieval, modification or deletion of rules. The rule server will also compile the condition

and action functions associated with a rule and archive them into libraries. At the lowermost layer is the file system where rules are persisted. A dynamic loader is an additional unit needed for finally bringing persisted rules into memory in order to load these rules on the GED by calling EVENT and RULE constructors at GED runtime.

Figure 11 shows the different phases of rule creation.

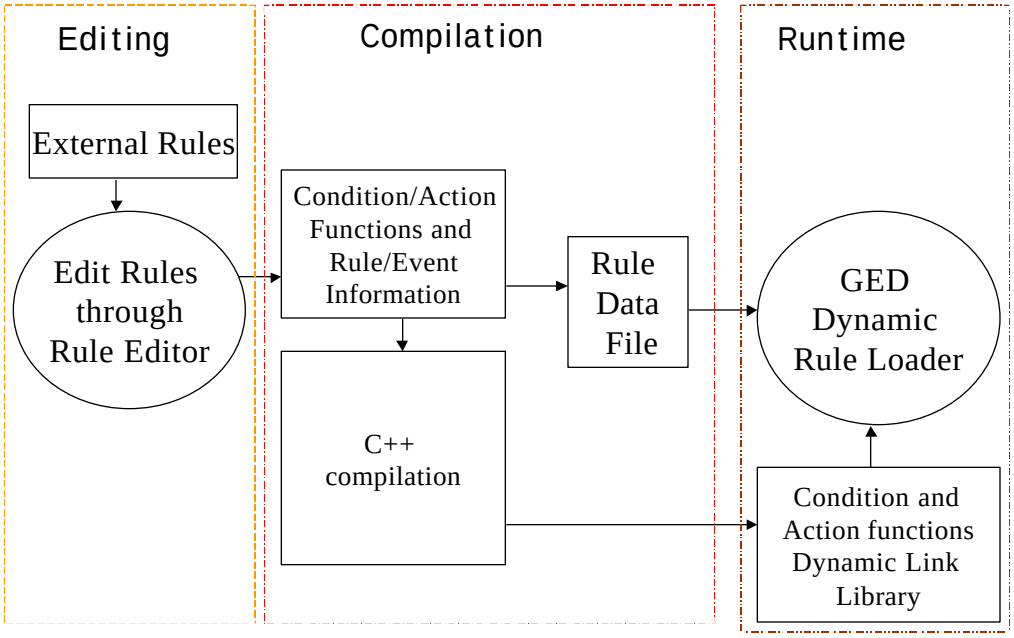


Figure 11: Phases of Rule Creation

6.3 Rule Persistence

Sentinel applications use Exodus Storage Manager to store and manage objects. Exodus is also currently used as the storage manager of the rule database, for persisting rules defined through the rule editor interface. Exodus storage manager stores all data on volumes which are either UNIX files or raw disk partitions. The Object Query Language (OQL) is used for accessing data stored in the database. Using Exodus makes the GED

dependent on OODB and Exodus. To make the GED independent of Exodus, rule information needs to be persisted separately in files by the rule server. Strings of information received by the rule server from the rule editor need to be written to files in a format that is efficient and convenient to read.

6.4 Dynamic Loading of Rules

In order to dynamically load pre-composed rules onto the GED, a function named *load_ged_rules* will be called at GED start up time before it does handshakes with new clients. This function retrieves the persisted rules, events, condition and action names. It then associates rules with their respective events and creates lists of event and rule objects in memory. It then traverses these lists and calls appropriate RULE and EVENT constructors for creating rule and event objects. On traversing the lists of rule objects it will read the condition and action names and invoke the dynamic loader to dynamically load the condition and action functions for the rules. The events and rules will finally be inserted into the G_GED global event graph. Details are explained in the next section.

6.5 Portability

The rule server needs to be portable and will thus be implemented in Java.

1. Java has security, network, and machine/ platform independent features.
2. Java is an object-oriented programming language with syntax is similar to C and C++ that are "main-line", industry-proven languages.
3. Java has been integrated with the Word Wide Web (WWW). By using Java as the language and by embedding Java programs in the form of applets into HTML pages, the program can also be accessed from the web.

The following chapter discusses implementation details of the modules for rule support.

CHAPTER 7
IMPLEMENTATION OF DYNAMIC RULE EDITOR SERVER

The previous chapter discussed design issues for the three-tired rule editor server architecture. This chapter describes the implementation details of its modules. Extensions that were made to the existing graphic interface to support global rules are discussed first. The interface communicates with the back end rule server by sending messages in ASCII in a certain format. Summary of the messages and their meanings are given next. The rule server was written in Java and is made up of *RuleEditor* class and *HandleConnection* classes. The purpose served by each class has been described. When new rules are defined through the interface, the server archives compiled condition and action functions into a library. When a rule is committed, rule information is also persisted into a rules data file. Format of the data file, followed by the details involved in reading the data file and construction of an event graph in memory to load rules on the GED at its runtime are the final sections of this chapter.

7.1 Extensions to the Rule Editor Graphic Interface

Abstract Window Toolkit, commonly referred to as the AWT is used for the graphical interface. The following 4 classes make up the functionality of the toolkit: *Component* class, *Container* class, *Graphics* class, and *LayoutManager* interface. The AWT is a platform-independent windowing toolkit. The AWT delegates the actual rendering and behavior of frames to the native platform-dependent windowing system.

In the extended interface the user is given a choice of defining rules on (1) Global

events or (2) local events. If the user chooses global he can either define his own group or choose from a set of groups already formed. Each group is made up of a set of applications. Files belonging to each group are in a separate directory in the file system. Once the group is chosen, the editor displays the global events belonging to the applications within the group. User can define rules on these global events. Rules can be defined on global primitive events or on composite events which the user will compose using the interface. The user, through the interface, also defines the condition and action functions that are sent to the back end rule server where they are compiled and archived into a library. Results of compilation of condition and action functions will be sent back to the interface where messages are displayed. The user can then reedit the functions or follow a sequence of windows to define rule name and rule specific information such as context, priority, coupling etc and finally commit the rule. When rule information is committed, it will be persisted by the back end server and will be reloaded at GED runtime.

7.2 Message Driven Services

In order to communicate with the back end rule server, the interface connects with the rule server on a specified host and port using a socket. Information from the editor interface to the rule server flows in the form of string messages over the socket. Following is the summary of messages that can be sent by the interface to the rule server to request a specific service:

"Ggroup\n" : This message asks the interface to send names of global events of applications belonging to the specified group. All files belonging to each group are placed in a separate directory given by the group name. The Rule editor server reads the

Global_signatures file belonging to that group to extract the appropriate global event information from it. It then sends this information in the form of strings to the editor, where it will be displayed in a menu window.

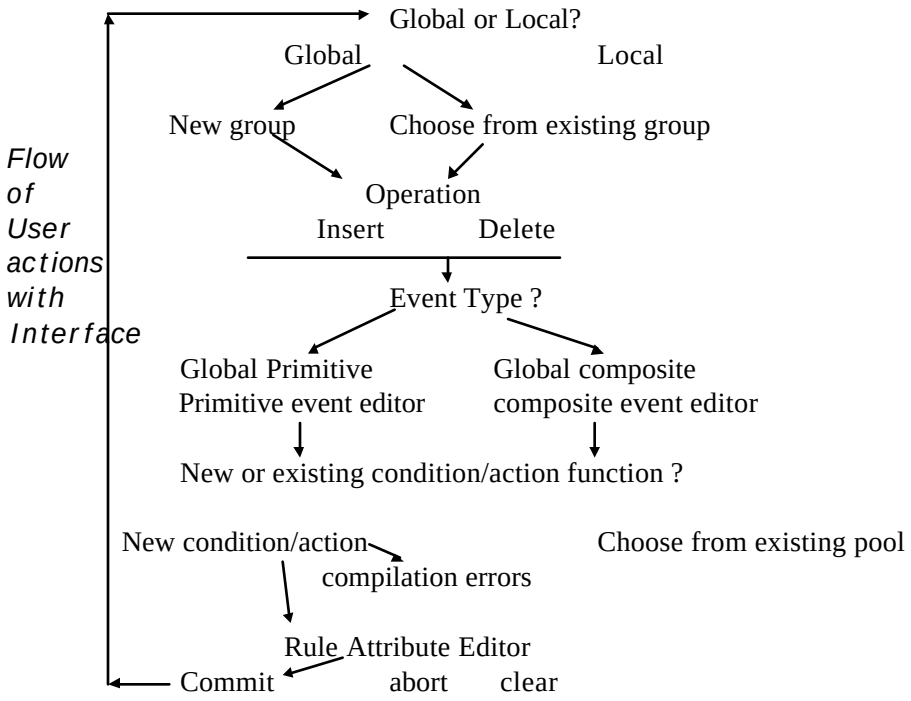


Figure 12: Flow of User Interaction with Interface

"$group\n" : Send the name list of the given group's existing condition functions to the rule editor interface. The information is read from the rules data file.

"~group\n" : Send the name list of the given group's existing action functions to the requesting client. The information is read from the rules data file.

"*compile_functions\n*" : Compile the given functions in the temporary directory of the given group and send the results of the compilations to the requesting client.

"commit\n" : Add the given condition/action functions to dynamic linking library (if they are new), move the source files of the functions to function pools belonging to the group (if they are new), and persist the given rule information in the rule log file.

"commit_delete\n" : Remove the given condition/action functions from the dynamic linking library space (if they are not shared), remove the source files of the functions to function pools belonging to the group, and remove the rule information from the rule log file based on the given type identifier.

"abort\n" : Clean up the given functions (both source files and object files) in the temporary directory.

7.3 Server Classes

The rule server is written in Java and has two main classes: RuleEditorServer class and HandleConnections class. **RuleEditorServer class** takes a connection request from the editor interface and then spawns a new thread to handle the connection. Since Java is a system independent language, the program cannot directly access environment variables. While starting the server, environment variables such as path of the sentinel system directory, which are used in accessing the various data files, must to be made available to the program. This is done by running a shell script, which loads the environment variables into the *System.properties* structure that can be accessed by the Java program. TCP/IP sockets are used for communication with interface. Socket object is an instance of *ServerSocket* class provided in the Java API. Once the interface establishes a connection with the rule server, string messages are exchanged between the two processes and processing is handled in the *HandleConnection* class.

HandleConnections class invokes the method *handle()* to service the requests

such as persist, retrieve, or modify rules which are sent by the interface. Based on the type of message the request is processed and results are sent back to the interface. To do system dependent tasks, such as use *make* on the Unix system, the server uses *exec* command to fork a new process that runs a Perl script to do the operations.

7.4 Rule Persistence

Since the definition of rules through the interface and the loading of rules on the GED server does not happen at the same time, rule data needs to be persisted by the server so that rules can be reconstructed at GED runtime. Whenever a rule is committed through the interface, information needed to construct the rule is written into a rules data file. Each record contains information of a single rule. Each record ends with a field called *Valid.* When a rule is deleted, the *Valid* field is set to 0. When rules are read from the file, records whose *Valid* field equals 0 are ignored. Following is the format of a record in the Rules Data file:

Opcode evnt_name send_back site event(type) con_evnt1con_evnt2 con_evnt3 rule_name priority coupling context trigger Valid

Opcode is a numeric value that gives the type of the global event. For example, a global primitive event has an *opcode* of 0, composite event *"and"* has an *opcode* of 1 and the composite event *"or"* has an *opcode* of 2. *event_name* is the name of event on which the rule is defined. *send_back* is a numeric value that determines whether the event given by *event_name* will be propagated back to the consumer. This field is always set to 0 in the rules data file as the EVENT node will only be constructed to fire a rule at the GED site and the event occurrence does not have to be sent to any consumer client. At a later stage when the GED is running, if a new client connects with the GED and registers for

the same event, then the value of *send_back* will be made to 1 in the EVENT node in memory. *Site* is the name of the *send_back* site. Its is given a dummy value as there is no consumer that has subscribed for this event at the time of Rule creation. If the event given by *event_name* is composite, then *con_event1* is the name of its first constituent event. *con_event2* is the name of second constituent event. *con_event3* is the name of the third constituent event in cases of events such as *A* or *A** or *NOT*. *rule_name* is the name of rule. Context, priority, trigger and coupling give the values of the respective parameters associated with the rule. *Valid* is a numeric field that determines if rule has been deleted.

7.5 Dynamic Loading of Rules on the GED

In order to make the GED capable of supporting rules, a RULE class was incorporated into the GED code. An object belonging to the RULE class inherits from NOTIFIABLE and REACTIVE classes in the GED class hierarchy. A RULE object is *notifiable* because it must be notified when the corresponding event occurs. At the same time, it is reactive because the action of the rule object may raise an event that triggers another rule, leading to nested execution of rules. When the GED is started, rules data that was persisted into the log files must be loaded into memory to create RULE and EVENT nodes that form the Global event graph. This is done by the *load_ged_rules()* function. To prevent the rule and event nodes from being duplicated, a main memory data structure (*EventList*) is created first from the data read from the file. Construction of the *EventList* structure in memory also makes the construction of composite EVENTs more efficient. It prevents the need to again search the rule data file for complete information about a component event that makes up a composite event. The rules data file has to be

read just once and all the information about the component events is made readily available in memory for the construction of EVENT nodes. Since every rule is related to a specific event, a *RuleNode* is inserted into the rule list of its respective *EventNode.*

The Following Classes were defined for the creation of graph in memory:

RuleNode contains rule information such as rule name, context, priority, trigger mode, condition and function names.

RuleList is the list of rule nodes. Each event node contains a list of rules to be fired on occurrence of that event.

EventNode contains event information such as event name, site name, component events (in case of a composite event) and a rule list, which is the list of rules defined on this event. Whenever a new rule is defined on this event, it is inserted into its rule list.

EventList is the list of Event nodes. Figure 13 gives the *EventList* data structure.

After reading each record from the file, a *RuleNode* is constructed. If the Invalid field of a record is 1 then no *RuleNode* is constructed for that record. If an *EventNode* does not exist for the corresponding event, then it is also constructed.

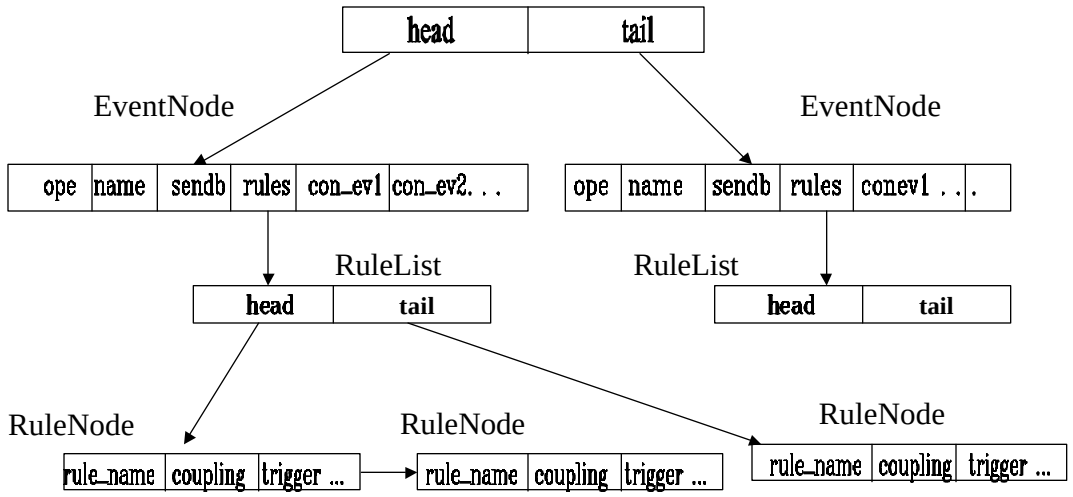


Figure 13: EventList Data Structure

To construct a composite *EventNode*, the *find()* operation is used which searches the *EventList* based on the constituent event name and returns a pointer to the constituent *EventNode*. Therefore each composite *EventNode* has pointers to its component *EventNodes.* After checking for duplicates, a *RuleNode* is inserted into the *RuleList* of the *EventNode*. New *EventNodes* are inserted into the *EventList*. After all records from the rules data file are read, construction of the *EventList* is complete. The *EventList* is then traversed for construction of the EVENT and RULE nodes, which make up the G_GED global event graph.

The library *GlobaldynCondActLib* contains the object code of the condition and action functions that had been compiled in the preprocessing and compilation phase.

While loading rules on the GED, this library is first opened (using *dlopen*) and the library handle is obtained. After the *EventList* is built in memory the function *create_events_rules*() is called on the *EventList* to construct EVENT and RULE nodes. The *create_events_rules* function takes the library handle as its argument. EVENT constructors are called for those nodes in the *EventList* that have a nonempty *RuleList.* While creating the EVENT nodes, the *config_name_list* is also checked. This list contains the mappings from the machine names used at compile time to the machine names used at actual GED runtime for the site which produces the event. Before creating any EVENT node, the G_GED graph is searched for a node with the same name by using the *get_prt_comp()* function. If the EVENT node is already present in the G_GED, it need not be constructed. This prevents creation of the same node multiple times and saves memory. For each *EventNode*, the *RuleList* is then traversed to create RULE objects which *Subscribe()* to the EVENT. While constructing the RULE object, *dlsym* call is made to the dynamic linking library. The *dlsym* call takes the condition or action function name to dynamically load the object code for the condition or action function into the memory. The following is the code from the body of "*create_rule"* method present in the *RuleNode* class. A RULE constructor is being called below:

```
int (*fptr_cond)(L_OF_L_LIST*)=NULL;
void (*fptr_act)(L_OF_L_LIST*)=NULL;
fptr_cond=(int(*)(L_OF_L_LIST*))dlsym(dyn_lib_handle,condition_name);
fptr_act=(void(*)(L_OF_L_LIST*))dlsym(dyn_lib_handle,action_name);
…..
RULE *rule_ptr = new RULE(rule_name, event_ptr, fptr_cond, fptr_act, coupling,
context);
……
……
```

In this way RULE objects are constructed and loaded on the G_GED. The

site_event_list is a list maintained at the GED that has one node per producer. Each node of the list contains the potential events that the producer must send to the GED. Every time an EVENT node is constructed for defining a RULE, the producer's list (*site_event_list)* has to be updated so that a producer connecting with the GED will know that it must send occurrences of this event to the GED. This is done by calling *update_site_evnt_list()* function. When GED handshakes with clients that are producers, they will read the *site_event_list* to know which events to send to the GED. For each event occurrence that is sent to the GED, the rule condition will be executed and if it evaluates to true, the rule will be fired.

CHAPTER 8
CONCLUSION AND FUTURE WORK

8.1 Conclusion

This thesis extends earlier work on the Global Event Detector in Sentinel to make the GED scalable and to improve its performance. A multithreaded architecture for the GED was proposed for increasing its performance and scalability. Synchronization issues related to multithreading were addressed. A lock hash table was implemented for handling locking of the Global event graph and read-write locks were used appropriately for locking the consumer event list that is also a heavily accessed data structure in the GED. Performance of the recoverable GED was also improved in logging and buffer management by implementing a better format for the log files and by changing the algorithm to deal with handling of events in the buffer. Performance measurements show that multithreading resulted in a reduction in the response time for the producer as well as consumers of events. This thesis also expressed the need for rule support and implemented a rule server that allowed the GED to fire rules on global events at its site.

8.2 Future Work

1. GED can be made hierarchical so that the GED can serve both as a server to monitor events for clients, and as a client that will register for events from other servers.

   We can have a hierarchy of GEDs to support real life applications where filtering of data is taking place. A localized group of applications can be at one hierarchy. The

hierarchy may reflect a common chain of organizations.

2. Replicated GED: Clients connect to more than one GED but send events to either one or both (depending upon whether the global event space is partitioned or replicated).

REFERENCES

[1] L. Hyesun, Support for Temporal Events in Sentinel: Design, Implementation, and Preprocessing. Master's thesis, University of Florida, Gainesville, 1996.

[2] V. Krishnaprasad, Event Detection for Supporting Active Capability in an OODBMS: Semantics, Architecture, and Implementation. Mater's thesis, University of Florida, Gainesville, 1994.

[3] S. Chakravarthy, V. Krishnaprasad, E. Anwar, and S.-K. Kim, Composite Events for Active Databases: Semantics, Contexts, and Detection. In *Proceedings, International Conference on Very Large Data Bases*, pages 606-617, August 1994.

[4] H. Liao, Global Events In Sentinel: Design and Implementaion of a Global Event Detector, University of Florida, Gainesville, 1997.

[5] E. Anwar, L. Maugis, and S. Chakravarthy. A New Perspective on Rule Support for Object-Oriented Databases. In *Proceedings, International Conference on Management of Data*, pages 99-108, Washington, D.C., May 1993.

[6] OODB. OpenOODB Toolkit, Release 0.2 (Alpha) Document. Texas Instruments, Dallas, September 1993.

[7] Hung-ju Chu, A flxible ECA Dynamic Rule Editor for Sentinel: Design and Implementation. Master's thesis, University of Florida, Gainesville, 1998.

[8] John Bloomer, Power Programming with RPC. O'Reilly & Associates, Inc, February 1992.

[9] Nichols B, Buttlar D & Farell J.P, Pthreads Programming: A POSIX Standard for better Multiprocessing, O'Reilly publications, 1st Ed, September 1996.

[10] Donald Lewine, Posix Threads programming guide. September 1994.

[11] Jennifer Sung, A Recoverable Asynchronous Event Manager for Supporting distributed Active databases. Master's thesis, University of Florida, 1998.

[12] S. Han, Three-Tire Architecture for Sentinel Applications and Tools: Separating Presentation from Functionality, University of Florida, Gainesville, 1997.

BIOGRAPHICAL SKETCH

Gauri Sukhatankar was born on October 1, 1973 in Mumbai, India. She received her Bachelor of Science degree in electrical and computer engineering from University of Florida in December 1996. In the Spring of 1997, she started her graduate studies in computer and information science and engineering at the University of Florida. She expects to receive her Master of Science degree in computer and information science and engineering from the University of Florida, Gainesville, Florida, in May 1999. Her research interests include Active and Object-oriented databases and Data Warehousing.