A RECOVERABLE ASYNCHRONOUS EVENT MANAGER FOR SUPPORTING DISTRIBUTED ACTIVE DATABASES

BY

JENNIFER C. SUNG

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

1998

Dedicated to my
family

ACKNOWLEDGMENTS

First and foremost, I would like to express my deepest sincere gratitude to my advisor, Dr. Sharma Chakravarthy, for giving me an opportunity to work on such an interesting and challenging Sentinel project and providing me great guidance and support through the course of this research work. I would also like to thank Dr. Eric Hanson and Dr. Herman Lam for serving on my committee.

I am also grateful to Sharon Grant for maintaining a well administered research environment and her commitment to work. Sincere appreciation is due to Hyoungjin Kim, Shiby Thomas, and Roger Le for their invaluable help and advice during the implementation of this work. I would also like to thank all my friends for their support and encouragement.

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

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

TABLE OF CONTENTS

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

A RECOVERABLE ASYNCHRONOUS EVENT MANAGER FOR SUPPORTING DISTRIBUTED ACTIVE DATABASES

By

Jennifer C. Sung

May 1998

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

An active Database Management System (DBMS), unlike the traditional DBMS, allows users to specify actions to be taken automatically when certain conditions evaluate to true without user intervention. Thus, it enhances the functionality of conventional database systems by supporting event-based rules. Once rules are declared, the DBMS is responsible for detecting the occurrence of the event, evaluating the condition when the event is signaled, and executing the action if the condition evaluates to true.

Most of the active Object-Oriented Database Management Systems (OODBMS) developed so far do not address event specification outside of their own address space. However, many applications in the real world are distributed in nature and hence require support for distributed computing. A distributed application is a collection of multiple and logically interrelated applications distributed over a computer network. A computer system, like any other mechanical or electrical device, is subject to failures. In addition to the above events occurring during normal execution, they can be affected by failures.

There are a variety of causes of such failure, including logical errors, system errors, system crash and disk failure. In addition to these failures, a distributed environment needs to also deal with the failure of a site, the failure of a link, and loss of message and network partition. Moreover, when we are dealing with a distributed application, it becomes much more complicated since several sites may be participating in its execution. The failure of one of these sites or the failure of a communication link connecting these sites may result in erroneous computations.

The work on the Global Event Detector only supported monitoring events that were distributed in multiple applications. Robustness of the Global Event Detector and event persistence issues were not addressed. However, it is important to address robustness and recovery issues since rules can be specified on events that occur in one or more applications, and in the presence of failures, events are not delivered making distributed applications prone to failure. Thus, in order to have a reliable event detection and propagation, we want a recoverable Global Event Detector that can be brought to a consistent state following various types of failures, can tolerate system failures, and can continue to provide services when it recovers from failures. Therefore, the motivation of this thesis is to design a recoverable Global Event Detector, which is robust to failures of producers and consumers and guarantees delivery of events.

CHAPTER 1
INTRODUCTION

An active Database Management System (DBMS), unlike the traditional DBMS, allows users to specify actions to be taken automatically when certain conditions evaluate to true without user intervention. Thus, it enhances the functionality of conventional database systems by supporting event-based rules. Once rules are declared, the DBMS is responsible for detecting the occurrence of the event, evaluating the condition when the event is signaled, and executing the action if the condition evaluates to true.

Most of the active OODBMSs developed so far do not address event specification outside of their own address space. However, many applications in the real world are distributed in nature and hence require support for distributed computing. A distributed application is a collection of multiple and logically interrelated applications distributed over a computer network.

A distributed application is subject to failures in many ways. There are a variety of causes of such failure, including logical errors, system errors, system crash and disk failure. In addition to these failures, a distributed environment needs to also deal with the failure of a site, the failure of a link, and loss of message and network partition. Moreover, when we are dealing with a

distributed application system, it becomes much more complicated since several sites may

be participating in its execution. The failure of one of these sites, or the failure of a communication link connecting these sites, may result in erroneous computations.

The work on the Global Event Detector only supported monitoring events that were distributed in multiple applications. It adopts the client/server model and uses the Remote Procedure Call and sockets communication. The Global Event Detector acts as a server and provides global event detection services to clients through Remote Procedure Calls and sending socket messages. However, robustness on the Global Event Detector and event persistence issues were not addressed in the previous work of the Global Event Detector. It is important to address these issues since rules can be specified on events that occur in one or more applications, and there should be no surprise that when a failure occurs, as the distribution of events is not sufficient to make the distributed application reliable.

There are several cases that explain the importance of these issues. First, if a consumer crashes, events that are detected by the producer while it was down will be lost if there is no event persistence. Hence, the consumer will lose the events while it was down. This may result in a buffer overflow and the server may fail. Second, when a producer recovers, events that need to be sent to the server once it has been detected by Local Event Detector will be lost if there is no mechanism to properly maintain the event list at the server. Third, the Global Event Detector can also face failures and should be able to reconstruct its state at the time of recovery. Moreover, when a site crashes due to a failure, the other sites should not be affected by it and should perform normally.

Thus, in order to have a reliable DBMS, we want a recoverable Global Event Detector that can recover to a consistent state following various types of failures, can tolerate client failures, and can continue to provide services after it recovers from a failure. Therefore, the motivation of this thesis is to design a recoverable Global Event Detector with the characteristics of robustness to failures of producers and consumers, guaranteed delivery of events, recoverability for the Global Event Detector, and persistence of events and their parameters.

CHAPTER 2
RELATED WORK

In this chapter, several research efforts on monitoring the behavior of distributed systems and fault tolerance and recovery mechanisms are discussed.

2.1 CORBA

One of the services provided by CORBA [1] is an event service. Event service supports producers that produce events, and consumers that process them via event handlers. The push model and pull model are supported as event notification models. In the push model, a producer of events takes the initiative and initiates the transfer of event data to consumers. In the pull model, a consumer takes the initiative and requests event data from a producer.

2.2 Schwiderski Thesis

In Schwiderski's thesis [2], she presents a strategy to monitor the behavior of distributed systems and proposes an approach to event-driven monitoring of distributed systems which provides the full functionality of event specification, event semantics, and event detection.

In this section, a few research efforts on fault tolerance and recovery mechanisms are discussed.

2.3 ARIES

ARIES (Algorithm for Recovery and Isolation Exploiting Semantics) [3] supports fine-granularity locking and partial rollbacks using write-ahead logging (WAL). The WAL-based system record all transaction into a log. The log can be considered as an ever-growing sequential file. Hence, the log becomes the source of ensuring that either the transaction's committed actions are reflected in the database despite various types of failures, or that its uncommitted actions are undone. ARIES assigns a unique *log sequence number* (LSN) to every log record when that record is appended to the log. The LSNs are in an ascending order, so only when a transaction's committed status and all its log data are safely recorded on stable storage, by forcing the log up to the transaction's commit log record's LSN, can it be considered complete. This allows a restart recovery procedure to recover any transactions that were completed successfully but whose updated pages were not physically written to nonvolatile storage before the failure of the system. This means that a transaction is not allowed to commit until the redo portions of all log records of that transaction have been written to stable storage. The LSN concept will avoid attempting to redo an operation when the operation's effect is already present in the page. It also will avoid attempting to undo an operation when the operation's effect is not present in the page. ARIES also introduced the notion of compensation log records (CLR) that record updates performed during partial or total rollbacks of transactions during both normal and restart processing so that it will make redo idempotent.

Another paradigm that ARIES introduced is that during restart recovery the first thing it does is analysis, which is to repeat history. During redo phase, ARIES, by repeating history is to essentially reestablishes the state of the database as of the time of the system failure. A log record's update is redone if the affected page's page_LSN is less than the log record's LSN. The next phase is the undo pass during which all loser transactions' updates are rolled back, in reverse chronological order. This is done by continually taking the maximum of the LSNs of the next log record to be processed for each of the yet-to-be completely-undone loser transactions, until no transaction remains to be undone.

To extend the capabilities of ensure persistence of events and recovery of the Global Event Detector, we only adopted the write-ahead logging and the log sequence number concept. We did not apply ARIES algorithm in full because we do not have an undo for our global event detection.

2.4 Shadow Page

Shadow paging [3, 5] is a crash recovery mechanism for which two page tables are maintained during the life of a transaction: the *current* page table and the *shadow* page table. When the transaction starts, both page tables are identical. The shadow page table is never changed during the execution of a transaction. The current page table may be changed when a transaction performs a write operation. All input and output operations use the current page table to locate database pages on disk. When the transaction commits, the current page table is written to nonvolatile storage. The current page table

then becomes the new shadow page table and the next transaction is allowed to begin execution. When the system comes back up after a crash, shadow page table will be copied into main memory and use it for subsequent transaction processing. If the transaction aborts, the current page table is simply discarded. This guaranteed that the shadow page table points to the database pages corresponding to the state of the database prior to any transaction that was active at the time of the crash. Thus, aborts are automatic and no undo operations need be invoked. The advantages of shadow paging over log-based techniques are that the overhead of log-record output is eliminated, and recovery from crashes is significantly faster (since no undo or redo are needed). However, the drawbacks are data fragmentation and garbage collection and is more difficult than logging to adapt to systems that allow several transactions to execute concurrently.

2.5 POSTGRES Storage System

POSTGRES storage manager [4] provides transaction management and access to database objects in a collection of modules. Unlike ARIES, it does not use conventional write-ahead log. Instead, recovery from crashes is essentially instantaneous due to the fact that there is no code to run at recovery time. This is achieved because POSTGRES has adopted a novel storage system in which no data is ever overwritten but rather all updates are turned into insertions. The storage manager allows a user to optionally keep the entire past history of database objects on a write-once-read-many (WORM) optical disk (or other archival medium) in addition to the current state on an ordinary magnetic disk.

Thus, it has a vacuum cleaner that moves archival records of magnetic disk onto an archival storage system. Moreover, POSTGRES DBMS avoided the large monolithic single-flow-of-control architectures that are prevalent today, and instead, it uses one that supports a collection of asynchronous processes with concurrently performing DBMS functions.

CHAPTER 3
SUMMARY OF SNOOP IN SENTINEL

SNOOP [9, 10] is an event specification language in Sentinel for specifying ECA rules. It supports temporal, explicit, and composite events in addition to the traditional database events. The Snoop preprocessor transforms the ECA rules specified either as part of a class definition or as part of an application to C++ codes.

3.1 SNOOP Flags

Several flags are provided by spp to facilitate the use of global event detection and rules execution.

- -s :

  SPP is integrated with OpenOODB preprocessor ppCC. -s flag invokes the spp preprocessor to convert ECA rule specification in Sentinel applications into C++ codes.

- -Lgen <*filename*>, -Luse <*filename*> :

  These two flags are for local event detector (LED). In spp, *filename* is an event definition file that is created for each application. This file contains all the events and rules definition in C++, which is translated by spp. This C++ code is then inserted into the main module of an application by spp. Since the main module of

an application can exist under a different path from other modules, it is difficult for the main program to get the event and rule definition file created from other modules. By using -Lgen and -Luse flags this problem is solved. -Lgen <*filename*> is to create *filename*, which contains the events and rules definition. -Luse <*filename*> is to specify the *filename* that is created by the –Lgen. There could be as many –Luse flags as needed.

- -ged

  This flag is to invoke the GED server and can connect a stand-alone application to distributed applications. Applications that do not involve global event detection should not give this flag and can run without this overhead.

- -Ggen <*filename1*>, -Guse <*filename1*>, -Gsend <*filename2*>

  These three flags are for global event detector (GED). In spp, global specification file, *filename1*, is created for each application that has defined global event definitions in its application. A global specification file contains global event information for the server to build the global event graph. -Ggen <*filename1*> defines the global event specification *filename1* that is generated by the application. This flag is used for each module that contains any global event definitions using SNOOP language. -Guse <*filename1*> provides the path and name of the event specification file generated by -Ggen flag. There could multiple –Guse flags and each is corresponding to a –Ggen. -Gsend <*filename2*> defines the final global event specification file, *filename2*, used by the application. This

filename2 is used by the main module to create the linked list of global event information before sending it to the server to build the event graph.

3.2 Event Definition

Events can be classified into two categories: local events and global events. Each local event and global event can also subdivided into primitive events and composite events. Hence, there are four types of events that can be identified in a distributed database system:

- Local Primitive Event

  Local Primitive Events are events that are predefined in the application. Local Primitive Events can be database events and temporal events. Database events refer to database operations to manipulate data, such as insert, delete, etc., and can be transformed into events using event modifiers (begin and end). Temporal events refer to specific points on the time line.

  Primitive_event ::= event *event_modifier* method_signature

- Local Composite Event

  Local Composite Events are composed of local primitive events and other local composite events conjunct with event operators.[10]

  Composite_event ::= E1 operator E2

- Global Primitive Event

  Global Primitive Events are events, could be primitive or local composite, that are defined with events that are defined and detected outside of the current application

and are referenced by the current application in a distributed database system. Since global primitive events are associated with events that are defined and detected outside of the current application, there must be some way to acknowledge that event has occurred. Three attributes are added to primitive event syntax to solve this problem. *App_name*, *Remote_event_name* and *Host_name* indicate the remote application ID, event name at the remote site, and the machine name where the remote application is running on respectively.

Global_primitive_event ::= *Remote_event_name*::*Host_name__App_name*

- Global Composite Event

  Global Composite Events are events that related to event occurrences from many sites (including the local site). At least one of the constituent events must be a global event. This constituent event can be presented as a global event name or a global primitive event specification.

  Global_composite_event : E1 *operator* E2
  (at least one of E1/E2 has to be a global event)

Please refer to [9, 10] for detailed descriptions on the BNF of the SNOOP language.

CHAPTER 4
SUMMARY OF EVENT DETECTORS

Database events correspond to the execution of methods; 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, a Local Event Detector (LED) [7] should not only be detecting primitive events but composite events as well. In a distributed computing system, event detection should monitor the behavior of events in a distributed environment. This requires a mechanism to detect events occurring not only at a local site, but also at other remote sites. To accommodate global event detection, some extensions are added to the LED. Global Event Detector (GED) [8] is responsible for detecting events from different applications in a distributed database environment. It recognizes the occurrence of events, collects and records their parameters, and passes it to application rule managers to trigger the action of ECA rules.

4.1 Local Event Detector

The purpose of the local event detector (LED) is to detect the occurrence of local primitive events and local composite events. Each application has a local event detector at its own site, and when a primitive event occurs, it is detected by the local event detector (LED) and the application waits for the signaling of rules that are detected in the specified

mode. Composite events are detected by using a sequence of primitive events detected according to the specified parameter context of the composite event. The architecture of LED is illustrated in Figure 4.1.

Primitive events are signaled by adding a *Notify* procedure call and appropriate calls for the parameter collections, which are inserted in wrapper method by Sentinel preprocessor into the application. Hence, whenever a primitive event is detected, the LED will traverse an event graph as shown in Figure 4.1.

The LED is an EVNT_LIST, which is a linked list of EVNT_NODE. Each EVNT_NODE corresponds to a unique REACTIVE class. Every method of a REACTIVE class is a potential primitive event. EVNT_NODE has a *begin_of* event list, *end_of* event list, and *subscribe_list. begin_of* and *end_of* correspond to a primitive event that should be raised at the *beginning* or at the *end* of the method, respectively. Subscribe_list is a list of composite events. This is added in addition to the original LED architecture for the purpose of checking global events. The leaf nodes of the event graph correspond to primitive events and from which composite events are constructed. Each node of the event graph has an event subscriber and a rule subscriber that record the related composite events and rules. Whenever a primitive event is raised, it will notify its subscribers, which are their parent nodes. The composite events will maintain the occurrence of its constituent event occurrences as part of their parameter lists, which are stored separately for each context relevant to the node. For details of LED, refer to [7].

Figure 4.1 Data Structure of LED

4.2 Extended Local Event Detector

In addition to detecting local events, the LED 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 LED will detect a global event only when it receives an event notification from the GED server. To accommodate global events, a REMOTE class is added to the class hierarchy in the LED. REMOTE class is a derived class of the EVENT class, and it represents global event objects. Figure 4.2 shows the event class hierarchy of LED. Each client has a LED at each own site to detect local events. An Extended LED (ELED) is added for the capability of detecting global events in every local site. Similar to LED, ELED is an instance of EVNT_LIST; however, unlike LED, all of the NOTIFIABLE classes that it points to are all **REMOTE** classes instead of PRIMITIVE classes. Also, each EVNT_NODE uses producer's application ID as its key instead of the class name. In other words, each node of the EVNT_LIST is related to a unique application, and each node has a list of ELIST which only points to a REMOTE object that contains only global event instances that are detected outside of this application. Therefore, a local composite event can be constructed from REMOTE nodes, Primitive Event nodes, and other composite event nodes.

Hence, whenever a producer detects a global event, the GED server will receive an event notification with its application ID and parameter list. Then, the GED server will notify the consumer and after the consumer receives the event, it will then notify its

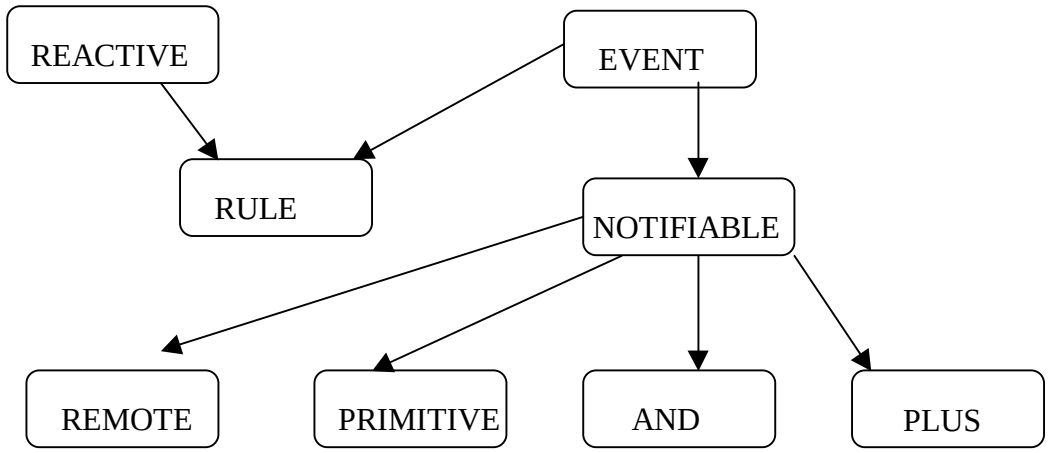


Figure 4.2 Event Class Hierarchy of LED

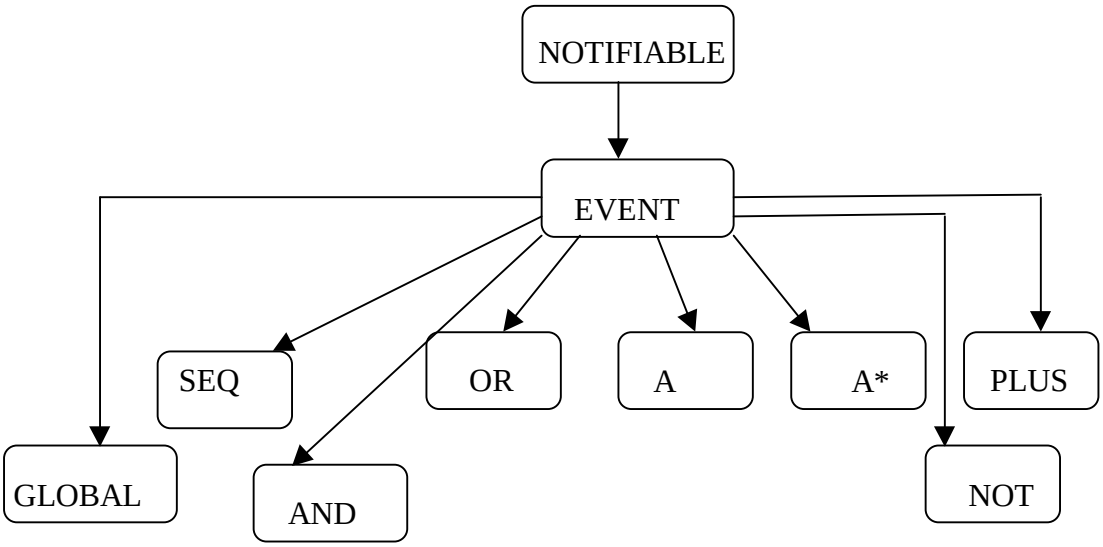


Figure 4.4 Class Hierarchy of GED

ELED. The ELED uses the application ID to determine which EVENT_LIST node and propagates the event notification to its corresponding REMOTE event instance. A REMOTE class works just like a PRIMITIVE class that the ELED will further notify its related composite events, which will be its parent nodes. Figure 4.3 demonstrate the structure of ELED.

4.3 Global Event Detector

The purpose of the global event detector (GED) is to detect events occurring not only at a local site, but also at other remote sites. Hence, the global event detector detects events that span several applications in a distributed database environment. It recognizes the occurrence of events, collects and records their parameters, and passes it to application rule managers to trigger the action of ECA rules.

4.3.1 Communication Architecture

Since each application has its own local event detector, a global event detector is to detect 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. Detailed descriptions on the GED alternative architectures are in [8]. 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. This event name list (cname_l) will be sent to the corresponding producer and

Figure 4.3 Data Structure of ELED

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. When an event is notified to the GED server, it sends a message to each consumer through socket. After the consumer has gotten this message, it makes a remote procedure call to the server and gets the event (event and its parameter list). Finally, the consumer traverses its ELED and the global event will be detected. The architecture is in Figure 4.4.

4.3.2 Global Event Detection

First, global primitive events are detected by their local event detectors at the corresponding remote sites. Then, event notifications are sent to the GED server. However, global composite events, unlike global primitive events, are more complex since the constituent events can be either local or global. The mechanism that is used to detect global composite events is to detect all the constituent local events of a composite event at its corresponding remote sites. Figure 4.5 illustrates this approach. In (A), the LED in S1 detects SOR and (B) the LED in T2 detects TAND. Once SOR or TAND got detected, S1 or T2 send the event to the GED server. Hence, G_AND will be detected in GED server in Figure 4.5 (C). By detecting global composite events at the local site will reduce the network communication overhead and improve the system performance compare to detecting global composite events at GED server.

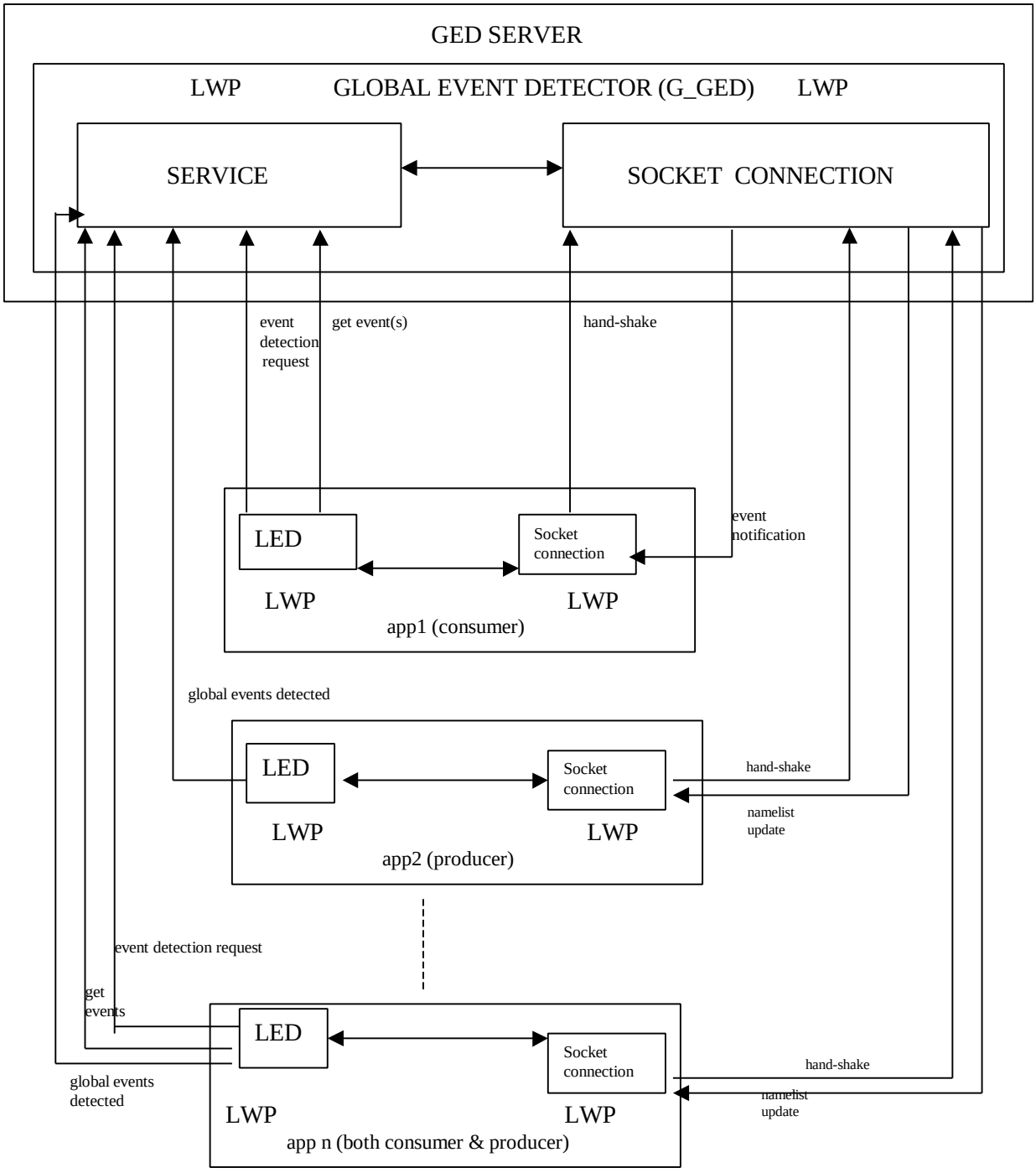
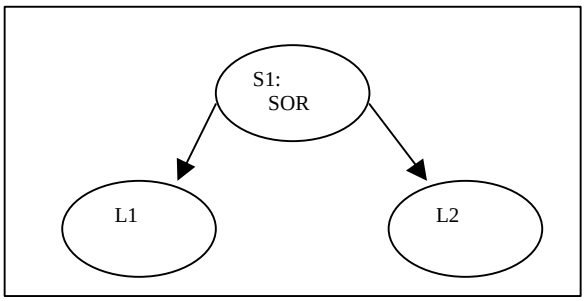
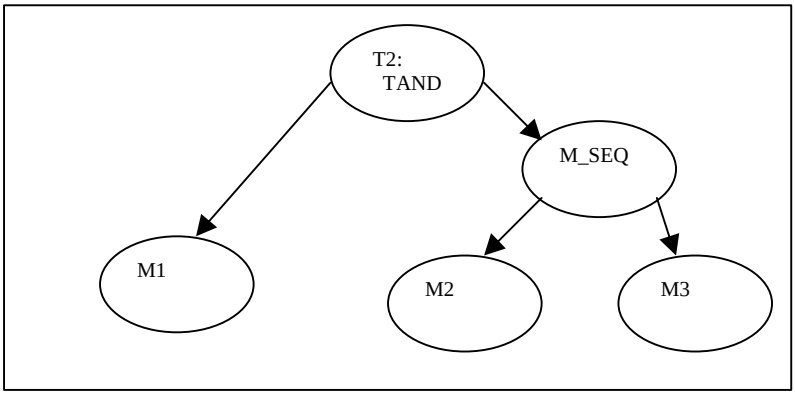


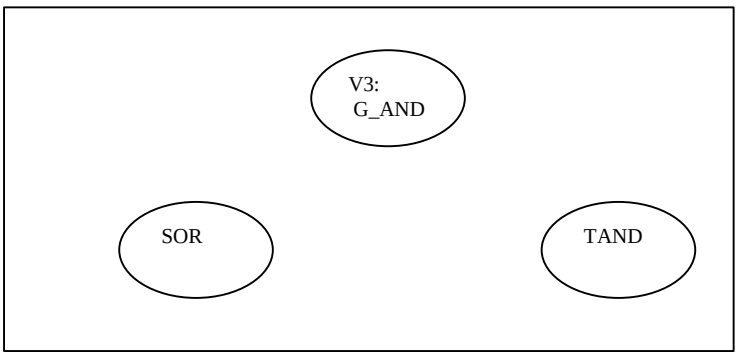
Figure 4.5 RPC and Socket Design Model of GED



(A) Application S1 : SOR is a composite event of L1 OR L2



(B) Application T2: TAND is a composite event of M1 ^ M_SEQ
M_SEQ is a composite of M1 << M3



(c) This is a global composite tree graph in GED server.
Application V3: defined an event G_AND = SOR ^ TAND.
SOR, and TAND are global events from application S and T.

Figure 4.6 Global Composite Event Detection

4.3.3 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 to external events that are detected outside of the local application. Therefore, we introduce the GLOBAL class 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 notified by the server after this event is raised. *send_ename* is the name of this event that is defined in *send_sname*. *event_no* denotes the instance number of the occurrence of this event. Figure 4.4 shows the class hierarchy of GED. Because of the possible time delay during communication and network failure, "P" and "P*" operators are not supported by GED.

Since the favor of reducing the network communication overhead to improve the system performance, we decided that global composite event is detected at the local site and a producer sends the event notification to GED server. Hence, 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 wanted. The information that a client sends is obtained from the global event specification file that is generated by *spp*. Refer to [8] for more information on global event specification file. Similar to the LED and the ELED, 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 a 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 compute its parameter link list. It has the similar concept as the LED and the ELED. Figure 4.7 shows the architecture of GED.

4.4 Summary of LED, ELED, and G_GED

First, whenever a client defines local primitive events or local composite events in its application, an event tree (LED) is built. In addition to primitive events, if a client also defines global events that are defined in remote sites, an Extended Local Event Detector (ELED) is built. Therefore, we have an Extended Local Event Detector (ELED) combined with a Local Event Detector (LED) at each client site to detect local events and global events that are wanted by remote sites. Then, we have a GED interface that allows communication between the GED server and clients that sends event detection requests and receives global event notifications. During the registration, the client's application ID is sent to the GED server, and the event specification file is sent to the GED server by a remote procedure call. By using the information in event specification file, a global event graph (G_GED) is built. So, the GED server receives an event notification from a producer whenever a global event is detected. Then, the GED server propagates G_GED, computes its parameter link list, and further notifies its parent nodes according to its subscribers. Finally, it sends this notification along with event's parameter list back to the consumer. In other words, the ELED then propagates and further notifies its parent nodes. Figure 4.8 shows the how these three coordinate with each other.

Figure 4.7 Architecture of G_GED

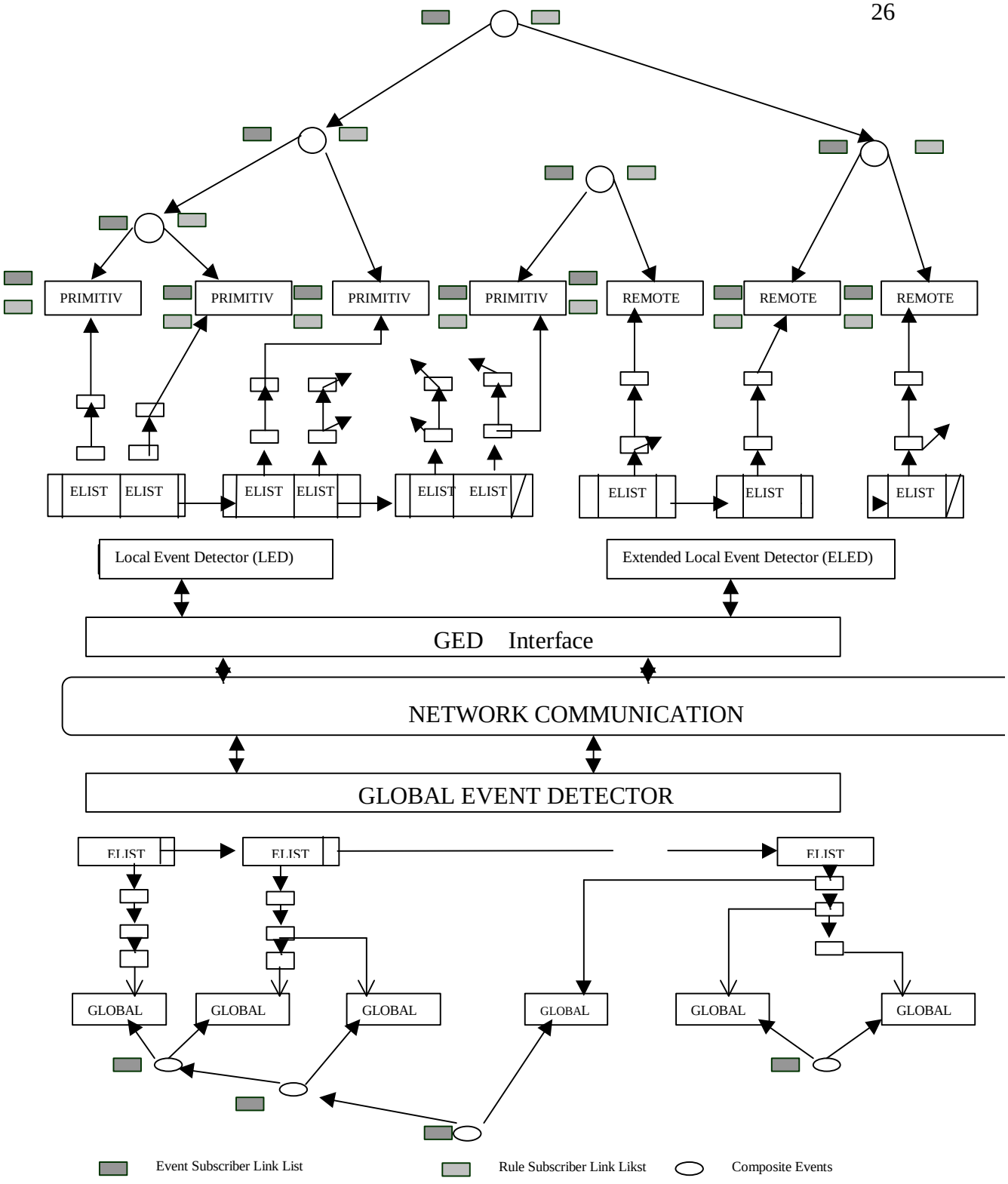


Figure 4.8 Data Structure of LED, ELED and G_GED

CHAPTER 5
DESIGN AND ALGORITHM

In this chapter, a design and algorithms for supporting machine independent on global event detection and a recoverable GED that can recover to a consistent state following various types of failures will be discussed.

5.1 Configuration File

As we discussed in section 3.2, the GED server identifies a global event by its producer's application ID (Hostname__appname), which is hard coded in the application. However, if for some reason that the producer is unable to run on the machine specified in the application, then we need to go back to the code, change the machine name, and recompile it. Therefore, by having a configuration file, we could make the producer application ID machine independent. Both old and new application IDs are specified in the configuration file, and the GED server will read both IDs and insert them to a linked list for mapping.

Mapping between the new and the old application ID is needed in several places. First, when a client registers with the GED server by using socket connection, it sends its application ID as a part of the message. Since the application ID is obtained from the

system call gethostname, it will get the name of the current machine where the client is currently running on. Hence, this application ID is the new application ID if it differs from the hard coded one. However, the GED server builds global event graph (G_GED) by using the global event specification file. Since the global event specification file is generated by spp, it contains the hard coded application ID. Therefore, mapping is needed whenever the GED server needs to access G_GED. This also applies to producer event list (site_evnt_list) since it reads the same data as G_GED. The other place where mapping is necessary is when reading or writing the event log file. Since the GED server receives the new application IDs from clients, it records everything under new application IDs. Event log files are named by using consumer application IDs. However, in this case, it would be useful to use the hard coded application ID instead of the new one because the user might change the machine where the application will be running on the next time. Therefore, by having this list, we are able to run applications on different machine without recompilation if desired.

5.2 Resume or Initialize Mode

If a client has disconnected from the GED server, the GED server must have a way to determine if this client wants to continue where it left off previously, or it wants to restart at the beginning when the client registers with the GED server the next time. To support this, we can add a flag at the command line when starting an application. When this flag is given, the client is to connect in a resume mode and sends a resume message along with its application ID, otherwise, it is to connect in an initialize mode.

The GED server determines the running mode of the client by the socket message. If the client is to run in an initialize mode, any previous log files must be deleted and must empty its buffer space so that events inserted previously do not get sent. On the other hand, if the client is to run in a resume mode, number of consumers must be calculated appropriately.

5.3 Robustness to failures of Producers

The GED server must maintain a list to keep track of events to be detected and sent to the GED server by a producer. This list is changed when a new consumer starts. Our original algorithm was to delete this list once the producer has received it. This causes a problem when a producer crashes and restarts. The producer can not remember the list when it restarts. Hence, the producer can not propagate global events properly. The algorithm to solve this problem is, first, not to delete the list when the producer receives it. Second, since the list can keep growing, the GED server must keep track of events that has been sent and events that need to be sent. We can add a pointer, which separates the events that have already been sent to the producer. Also, an attribute is added to the client address list, and the GED server sets this attribute when a client reconnects. Hence, the GED server will send the whole event list if this attribute is set.

5.4 Robustness to failures of Consumers

When a consumer registers with the GED server, it sends its global event specification file to build the global event graph (G_GED). Therefore, a consumer only needs to build the global event graph when it is connected in an initialize mode. In other

words, when a consumer reconnects in a resume mode, it does not need to rebuild the global event graph since it has already been built previously. This can be accomplished by using the same flag specified in section 6.2.

5.5 Guaranteed Delivery of Events

Since our communication protocol uses the non-blocking communication, when a consumer has crashed, its producers and the GED server cannot determine its state. Therefore, producers still send events to the GED server, but the consumer will not receive them. Hence, we need a mechanism to guaranteed delivery of events.

Write-ahead log files are adopted to ensure persistence of events and recovery of the event manager. Information is saved in a file before the data structures are modified. In addition to write-ahead log, log sequence number (lsn) is also used. LSN is assigned in ascending sequence and every event has a unique lsn. By doing write-ahead log and using log sequence number, the information in the log is used in restoring the state of the consumer. Hence, the GED server can send all pending events obtained from the log file when the consumer reconnects. The GED server can also recovers to its original state in case of a crash by obtaining from log files.

The GED server inserts a global event in a log file before it has inserted the event in the consumer list. An event log file contains two other lsns. One indicates that the consumer has received events up to this number, and the other indicates that the buffer contains events for this consumer up to this number.

5.6 Mutex Locks

Since the GED server creates a thread when it receives an event notification from the producer, there could be many applications that want to access the same data structure. Therefore, we used mutex locks to synchronize access to shared data structure. When the GED server recovers from a crash, it issues a lock that locks the entire recovery process, and releases the lock only when the recovery is over. This is to ensure that others can not access to the share data during the GED server recovery.

5.7 Buffer Management

As we discussed earlier, when a consumer has crashed, its producer cannot know its state and will still send events to the GED server. The GED server also cannot know that the consumer has crashed due to the non-blocking communication protocol. Hence, the GED server keeps insert events in the consumer event list. This may result in a buffer overflow and the GED server may fail. Therefore, we need a mechanism to mange the buffer pool for storing events.

There are two situations when the individual buffer is full. One is when a consumer cannot consume events fast enough, and the other is when a consumer has crashed. We propose an algorithm that there is a maximum number of events that the GED server can hold in its main memory. Each consumer is allowed to have a number of events in its buffer by dividing the maximum number of events that the GED server can hold by the number of consumers that are connecting to the GED server. We also assign a time out variable that helps to indicate the consumer's status. Every time an event cannot

be inserted to the list , the GED server updates a consumer's counter. When this counter has reached the time out limit, the GED server assumes that this consumer has crashed.

There are several cases when the GED server needs to insert an event to the buffer. First, during the normal execution where there are no new consumer registers with the GED server and no consumer has crashed, and if the buffer for this consumer is not full, then the GED server inserts the event to the list. If the buffer is full, it updates the consumer's counter. Second, if a new consumer has connected with the GED server, the number of events that each consumer is allowed to have will be smaller than the previous number. Hence, if the buffer is not over the new limit, then insert the event to the list. However, if the buffer is over the new limit, the GED server deletes the excess buffer space and updates the event log file and the consumer's counter. If a consumer has crashed, the number of events that the consumer is allowed to have will be larger than the previous one. So, if the consumer has no pending events in the log file, the GED server just insert the event to the list. If the consumer has pending event in the log file, and since there is more buffer space to hold events, the GED server inserts pending events to the list until either the buffer is full or there are no more pending events in the log file. Third, if the GED server has assumed that the consumer has crashed, it deletes the event list if it is not NULL.

CHAPTER 6
USAGE OF THE GED SERVER AND APPLICATONS

First of all, if global event detection is requested, we need to start the GED server before starting any client that participates in global event detection. (If an application that is neither a consumer nor a producer, it could run independently without the GED server.) The usage of the GED server is as follows:

- ged_server [-cfile path_name] [-nopersist]

  -cfile <path_name>

  This flag specifies the path name (directory) of the configuration file. If this flag is not given, the GED server gets the path name from the environment variable. The purpose of specifying the path name of the configuration file is that if multiple GED servers are running, each server will be able to read a different configuration file since each server could have a different port number and could have different clients connecting to it.

  -nopersist

  This flag indicates that the GED server will not be recoverable. In other words, no log files will be generated for recovery purpose. If this flag is not given, the default is to persist, which means that log files will be created under the specified directory in the configuration file.

The usage of running a client application that involves global event detection is as follow:

- <*application_name*> <*GED_machine_name*> <*global_event_specification_file* >
  [-resume] [-port port_number]

  application_name is the name of the application and is the first argument in the command line. *GED_machine_name* is the second argument and is the machine name where the GED server is running. *global_event_specification_file* is the third argument in the command line and is the global event specification file generated by spp for that application.

  resume

  This flag is to run the client in a resume mode, which means that the client has crashed previously and is restarting. This means that the GED server will treat the client as an existing one and will send all the information about what events to propagate to the client. If this flag is not given, the client will start in an initialize mode as if it were a new client. The purpose of having this flag is to have the ability to recover after a client has terminated abnormally. On the other hand, we would also like to have the ability to start fresh if some changes have been made to the client.

  -port port_number

  This flag specifies the port number of a GED server. If multiple GED servers are running, we need to specify which GED server does the client wants to connect with. By giving this flag, the above requirement is satisfied.

CHAPTER 7
IMPLEMENTAION DETAILS

In chapter 5, we proposed the design and algorithms to support machine independent on global event detection and a recoverable GED that can recovers to a consistent state following various types of failures. In this chapter, implementation details on those design and algorithms are discussed.

7.1 Implementation of a Configuration File and a Configuration List

When we start running the GED server, the first thing it does is to read a configuration file, which is named Global.config, from the specified directory. The purpose of having a configuration file is to make the producer application ID machine independent. The configuration file contains the old application ID and the new application ID and the GED server will do the mapping at run time. In addition to the mapping of an old application ID to a new application ID, other run time values can be supplied into the configuration file. For instance, a log directory where we want to keep the log files and a port number for the server are specified in the configuration file, so that we could have the capability to run multiple GED servers. The format of a configuration file is as follows:

::log_dir /cis/database15/sentinel/OpenOODB.1.1/Sentinel0.9/sentinel/

::mapping juice__app1 eagle__app1

::port 6000

The translation of the above configuration file is to create and save all the log files into /cis/database15/ sentinel/OpenOODB.1.1/Sentinel0.9/sentinel/. juice__app1 is the hard coded (old) application ID, eagle__app1 is the new application ID, and port number for the GED server is 6000.

The config_list (configuration list) table is a linked list that stores the old (hard coded) application IDs and the new application IDs. Figure 7.1 shows the data structure of the config_list. Mapping between the old application and the new application is accomplished by using this list. Log directory and port number are stored in two global variables.

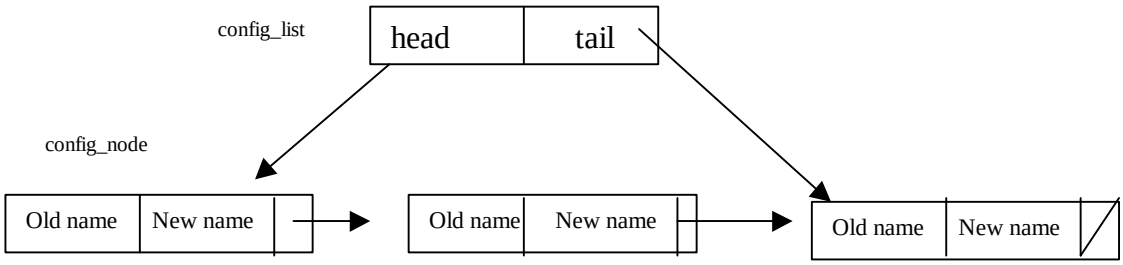


Figure 7.1 Data Structure of config_list

7.2 Implementation of the Registration Message

When a client registers with the GED server, it sends its application ID through the socket. However, the GED server needs to know if the client is running in a resume mode or an initialize mode. This can be implemented by adding a character and "::" to the beginning of the list. The "::" separates the keyword and the application ID. Therefore, if the client is running in a resume mode, the message will be r::application_ID, and the message will be i::application_ID if the client is running in an initialize mode.

7.3 Implementation of Robust Producers

When an event is detected by a Local Event Detector at its own site, the producer needs to know if this event has to be sent to the GED server so that the GED server can further notify the consumer that a global event of interest has been raised. Hence, when a consumer makes a Remote Procedure Call with evnt_decl_l, which contains information in the global event specification file, as its parameter, the procedure updates the global event graph and the consumer event list using the event_decl. This is done when a new consumer starts. The site_evnt_list (consumer event list) is an instance of SITE_EVNT_LIST_STR, which is a list of SITE_EVNT_NODE_STR. SITE_EVNT_NODE_STR contains *site_name,* and *name_list,* which correspond to the producer's application ID and a list of event names. There were originally two pointers that point to this list of event names. Our algorithm is to add a third pointer, *start*, which separates the events that have already been sent to the producer.

start points to events that stand for the starting point where this event name list

should be sent to the producer. After the event name list has been sent to the producer, *start* will be pointing to the last node of the list which will be NULL. Figure 7.2 shows the data structure of the producer event list. In Figure 7.3(a), shows the order of clients that register with GED server with respect to the time line. Figure 7.3(b) to (e) show how *start* pointer is managed.

We also add another attribute, *recovery_flag*, to the client address list (cli_addr_list). The GED server sets the *recovery_flag* to 1 when a client has reconnected with the GED server. This is accomplished by checking the application ID in the cli_addr_list. When the reconnecting client is a producer and its *recovery_flag* is set to 1, the GED server moves the *start* pointer to the head of the event list. Hence, when the producer reconnects, it receives all of the events in the event list. Therefore, By having the *start* pointer and the *recovery_flag*, the producer will not lose any events that are interested by its consumers. Figure 7.4 illustrates the data structure of cli_addr_list.

7.4 Implementation of Log Files

Information that is written in the event log file, in the case for client recovery, are global events. Whenever the GED server receives a global event from a producer, before it inserts it into the consumer event list (event_para_list), the GED server saves the global event into the log file. So, even if the client has crashed, it can be in a consistence state when it comes back using the log files created by the GED server. If all the global events are written into only one log file, then when a consumer has comes back up only a few seconds after its crash, the consumer has to read the entire file even if it has only a small

number of records in the file. This could cause a lot of overhead. Therefore, the alternative is to have multiple physical files, which will ease the job of archiving log records. In our design, we have a log file for each consumer since guaranteed event delivery is applicable only to consumers. The file name for each event log file is the consumer's old application ID with .log as its post-fix. The information written in an event log file is only the information about global events. This information only contains the consumer application ID, producer application ID, event name, and its parameter list. With only this information, the GED server can not know up to what point has it received the events, and hence cannot recover after a crash.

We have adopted the LSN concept in ARIES [3] into our design to solve this problem. The LSN concept is that every log record is assigned a unique log sequence number when that record is appended to the log. A log record corresponds to a global event that the GED server receives. The LSNs are assigned in ascending sequence and is a global variable in the GED server, which is called *event_counter*. It is initialized to 0 when the GED server starts, and it is incremented by 1 every time a global event is written into one of the event log files along with the global event information. Every time a consumer receives a global event, and before the GED server deletes the event from event_para_list, the GED server writes the last lsn from the consumer event list to the beginning of the log file. This LSN (*dlsn*) indicates that the consumer has received global events up to this

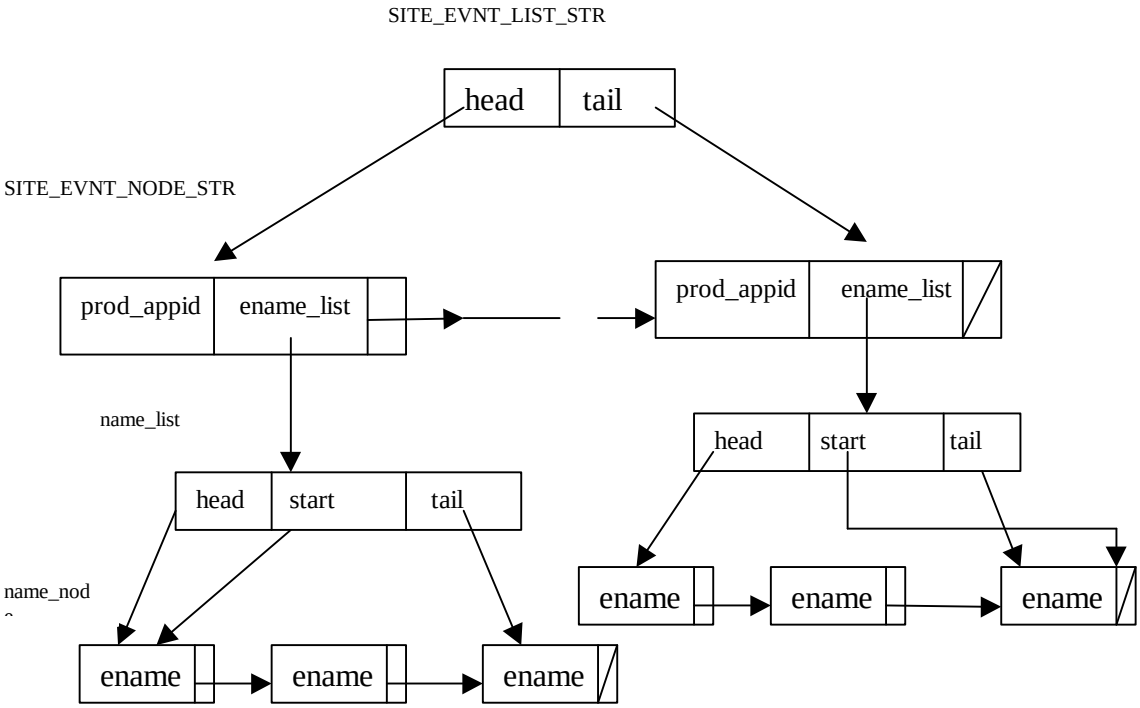


Figure 7.2 Data Structure of site_evnt_list (producer event list)

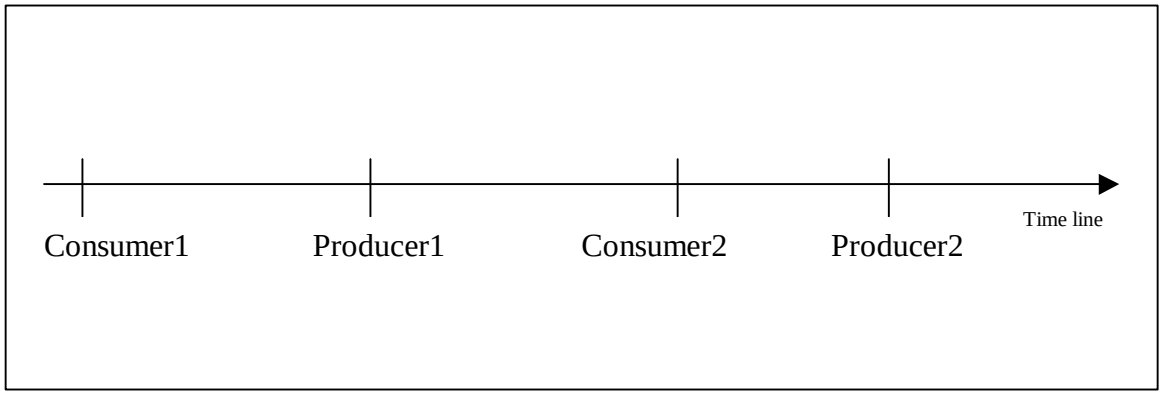


Figure 7.3 (a) The Order of Clients that registers with the GED server

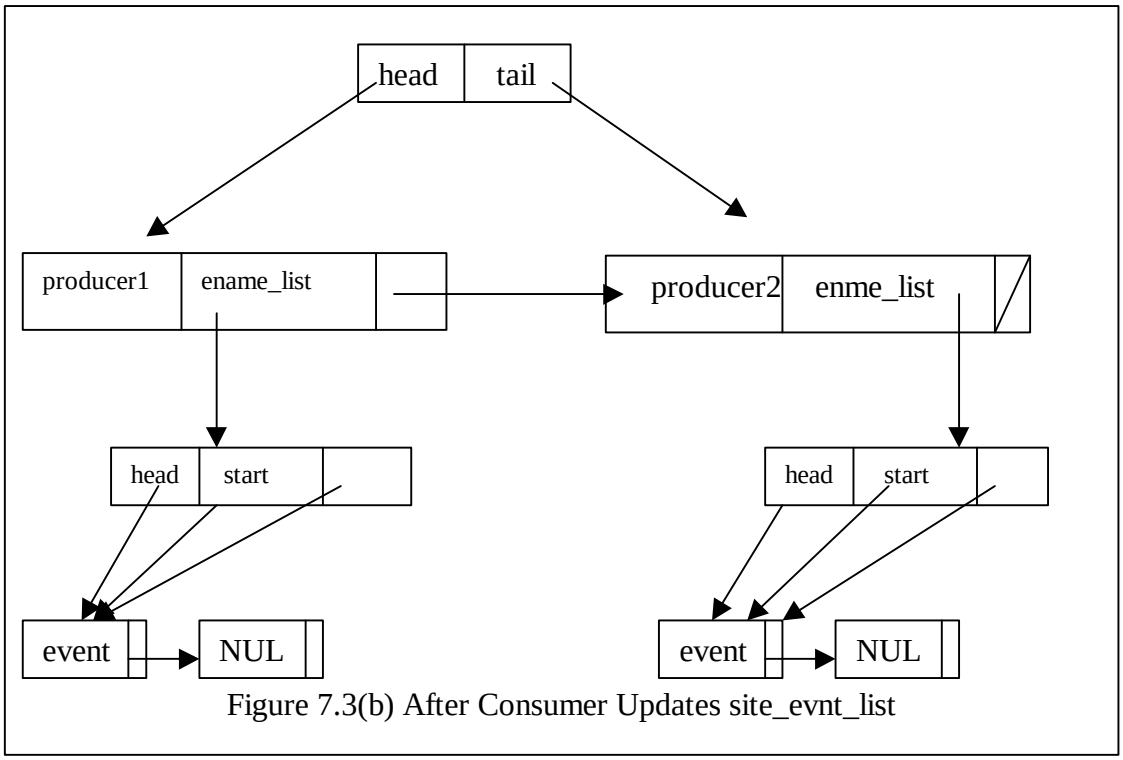


Figure 7.3(b) After Consumer Updates site_evnt_list

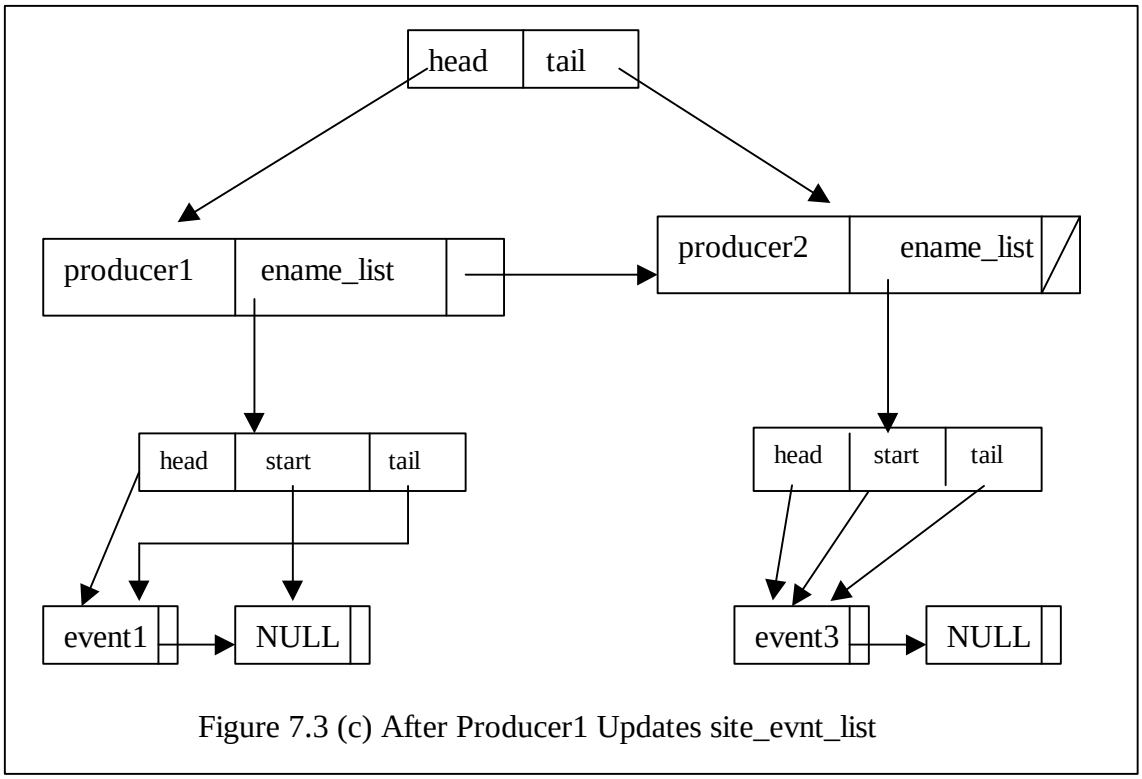


Figure 7.3 (c) After Producer1 Updates site_evnt_list

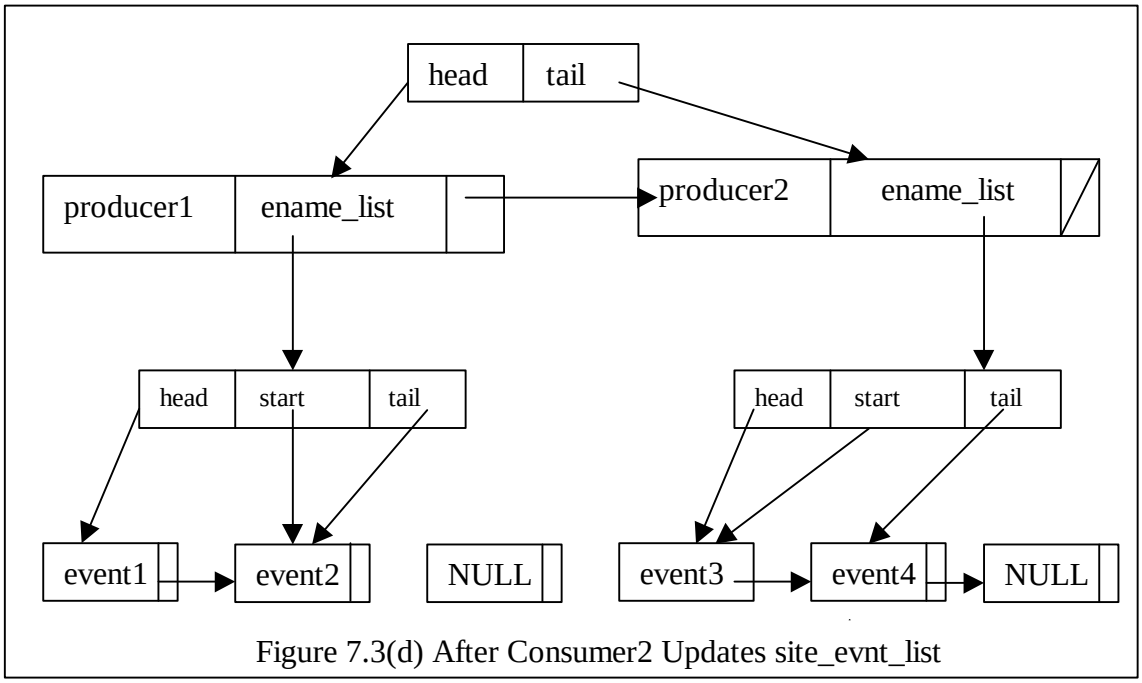


Figure 7.3(d) After Consumer2 Updates site_evnt_list

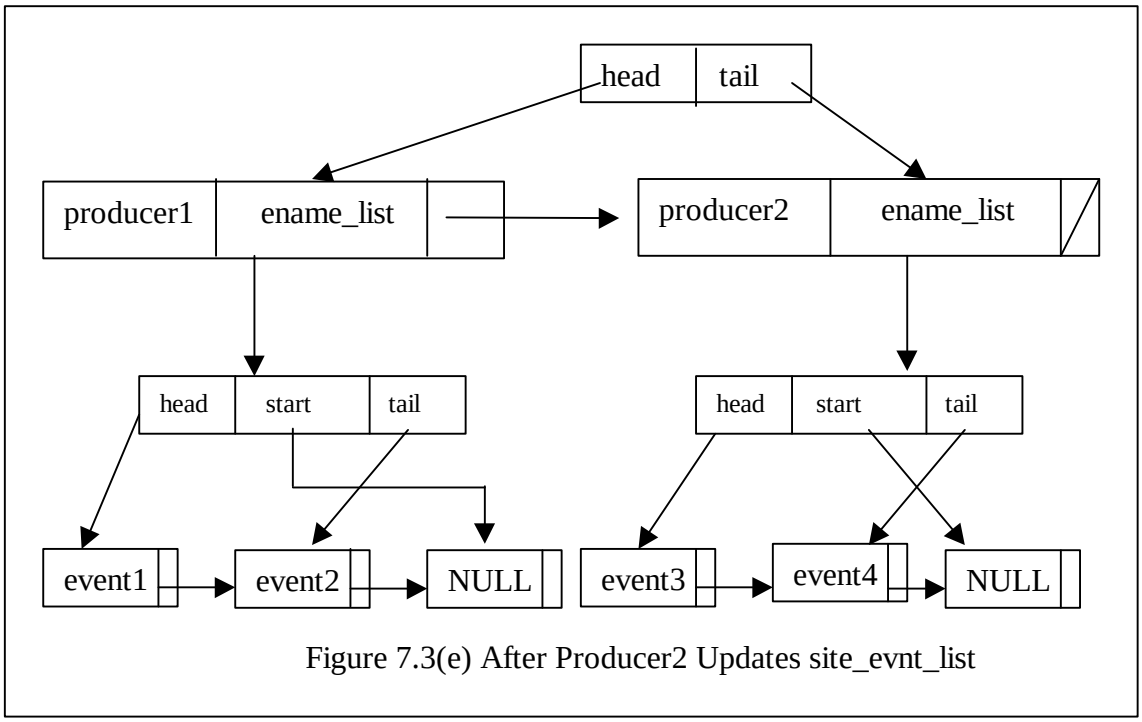


Figure 7.3(e) After Producer2 Updates site_evnt_list

Figure 7.3 Example of How site_evnt_list is Managed

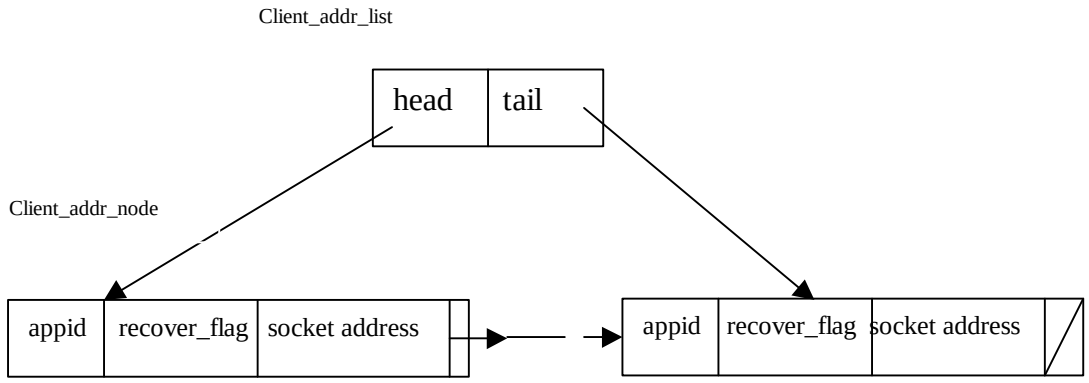


Figure 7.4 Data Structure of cli_addr_list (client address list)

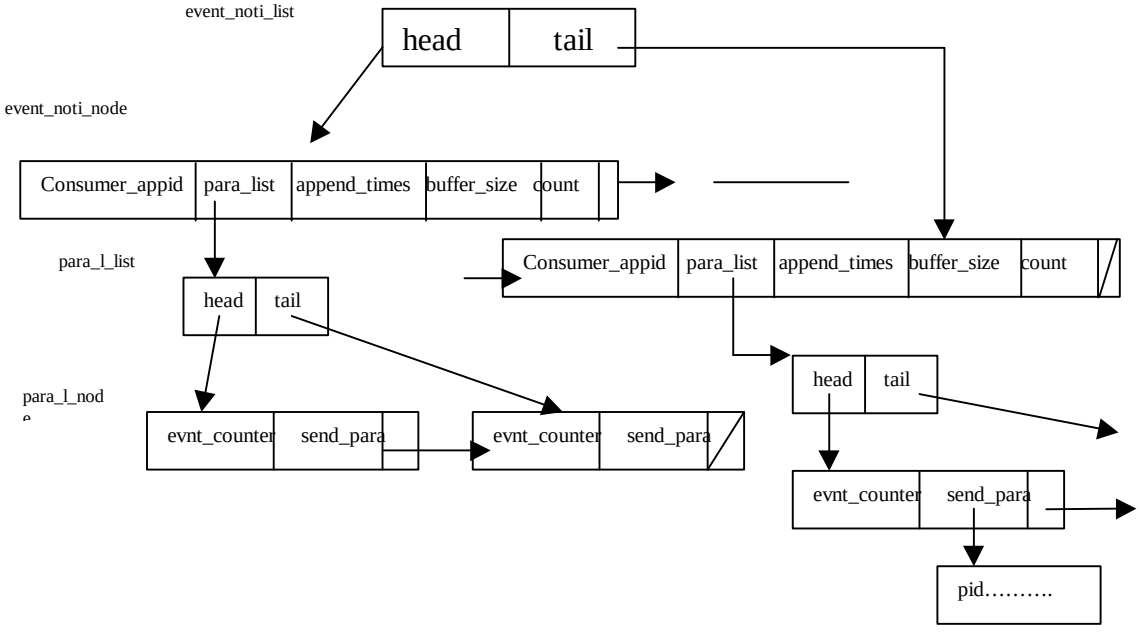


Figure 7.5 Data Structure of event_para_list (consumer event list)

number. If the consumer has crashed, the GED server still writes the *event_counter* and global events in the event log file. However, when the consumer recovers, the GED server knows up to what point has the consumer received the events before its crash by reading the *dlsn*. See Appendix for the format of each log file.

Information that is written in the log, in the case of the GED server recovery, is information to rebuild client address list (cli_addr_list), global event graph (G_GED), and producer event list (site_evnt_list). Therefore, we need to persist all information that is used to build these tables in log files. When a client registers with the GED server, the GED server receives the application ID and its socket ID. Before the GED server inserts in cli_addr_list, it writes application ID and socket ID into the log. The socket ID is written by using the sa_data[14] in struct sockaddr that is defined in sys/socket.h. This sa_data is a char array up 14 bytes of protocol-specific address. The contents of the 14 bytes of protocol-specific address are interpreted according to the type of address [11]. The GED server creates this log file as client_addr.log if it does not already exists, and the data is written in append mode. After the client has registered with the GED server, it will call the remote procedure global_notify and this is where the G_GED and site_evnt_list are built by using evnt_decl_l. Therefore, for the GED server to be able to rebuild these two lists when it recovers, it needs the data in evnt_decl_l. Hence, the GED server writes the data in evnt_decl_l in a log file named GED_Spec.log before it inserts them into G_GED or site_evnt_list. The GED_spec.log is also written in append mode.

7.5 Implementation of Recovery Lock

There could be multiple clients that connect to the GED server, and could be running concurrently. Locks are used to synchronize access to shared data structure. In the GED, there are five locks: *mutexPtr_Addrlist, mutexPtr_paralist, mutexPtr_site_evntlist, and mutexPtr_eventfile,* used on access to cli_addr_list, event_para_list, site_evnt_list and event log files respectively. The fifth lock is *mutexPtr_recoverlock,* which is used when the GED server recovers from a crash. All locks are exclusive locks, which means that nobody else will be able to access the data until the locks have been released. All locks will be released right after the operation has finished. *mutexPtr_recoverlock* is used when the GED server recovers. It is to lock the entire recovery process and release the lock only when recovery is over so that the GED server can send and receive events in the correct order. This is to ensure that others can not access to the share data during GED server recovery. Below is the pseudo code for recovery lock algorithm.

```
When GED Recovers :
    Obtain Recovery Lock
        client address recovery

        global event graph recovery

        event recovery

    Release Recovery Lock
```

```
When server accesses to the share data
structure:

    If (recovery lock is available)
        Does not obtain the recovery lock,
        but obtains the individual lock
       on the data structure
     Else
        Wait until the recovery lock is
         available .
```

7.6 Implementation of Buffer Management

The algorithm for buffer management that we used is that, first, we introduce a second variable *blsn*, which is written to the event log file. This *blsn* is placed after the *dlsn* that we described in section 6.3. When the GED server appends an event in the event log file, it also writes the *event_counter* to the *blsn* field. This indicates that events have been inserted in the buffer up to this number *(blsn)*. We also assign a maximum number of events *(BUFFER_MAX)* that the buffer can hold, and calculate the amount of buffer size that each consumer is allowed to have. The equation is *Each_buff_size = BUFFER_MAX / number_of_consumers*. *number_of_consumers* is added by 1 every time a consumer has registered with the GED server**.** Hence, each consumer has a buffer size (Each_buff_size) that each can hold, and we also extend the data structure of the event_para_list (consumer event list) to accommodate buffer management.

buff_size, count, and append_times, are added into the data structure of event_para_list, which indicate the previous buffer size, how many events are in the buffer, and how many appending events are in the log file after its buffer is full respectively. We also assign a number to the *TIMEDOUT* variable. This variable helps to indicate the status of the consumer. Every time the consumer's buffer is full, the GED server adds its *append_times* by 1 until the *append_times* is greater than the *TIMEOUT* VARIABLE. When the *append_times* is greater than the *TIMEOUT* variable, the GED server assumes that this consumer has crashed and deletes its event list (para_l_list) from the buffer. Figure 7.5 shows the data structure of event_para_list (consumer event list). Therefore, if

the reason that the buffer is full is due to slow consumer, as long as it does not reach the *TIMEOUT* point, the consumer will eventually receive them. If the consumer has crashed, it will eventually reach the point where *append_times* is greater than *TIMEDOUT*, and the GED server will free up the buffer space. The GED server will assume that the consumer has crashed and will not insert any more events to the buffer for this consumer. *number_of_consumers* is subtracted by 1 and the *Each_buff_size* is recalculated. Hence, *Each_buff_size* is greater than the previous one since the *number_of_consumers* is smaller than the previous *number_of_consumers.* Therefore, when a consumer compares the *Each_buff_size* with the *buff_size*, if the *Each_buff_size* is greater than the *buff_size*, then the GED server checks if there is any event that is pending in the event log file. If there is, then insert those events either until the buffer is full or there when there is no more pending event.

If a new consumer connects with the GED server, the GED server adds 1 to the *number_of_consumers* and recalculates the *Each_buff_size.* In this case, the *Each_buff_size* is less than the previous one. Hence, if the buffer for the consumer is not over the new limit, then insert the event to the list. However, if the buffer is over the new limit, then the GED server deletes the excess buffer and adds 1 to *append_times*.

Below is the pseudo code of Buffer Management Algorithm.

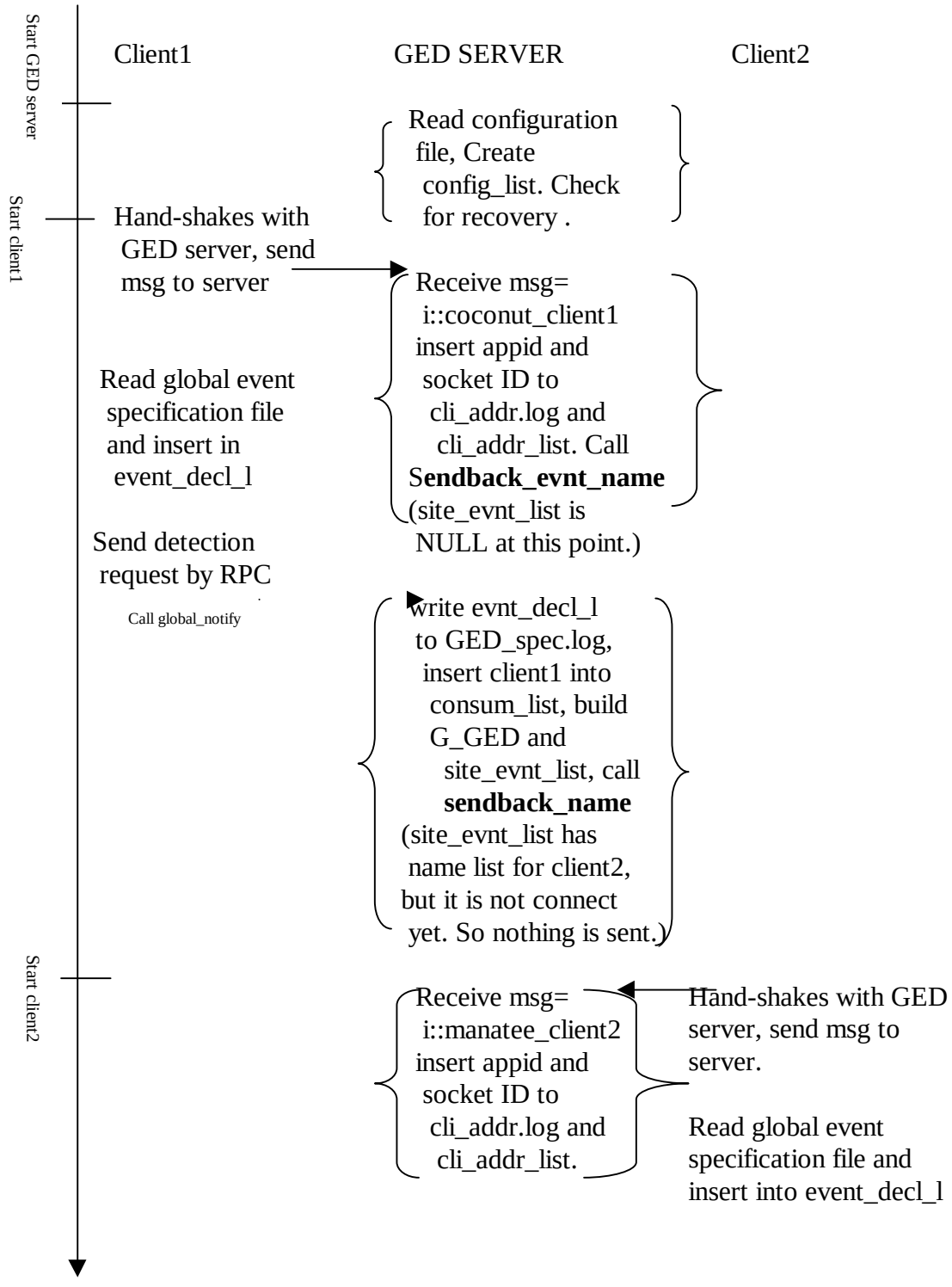
```
if (the consumer has not crashed)
        {
            if ( no consumer has came in and no consumer has crashed)
                if (buffer for the consumer is not full)
                        Insert the event to the buffer;
                else
```

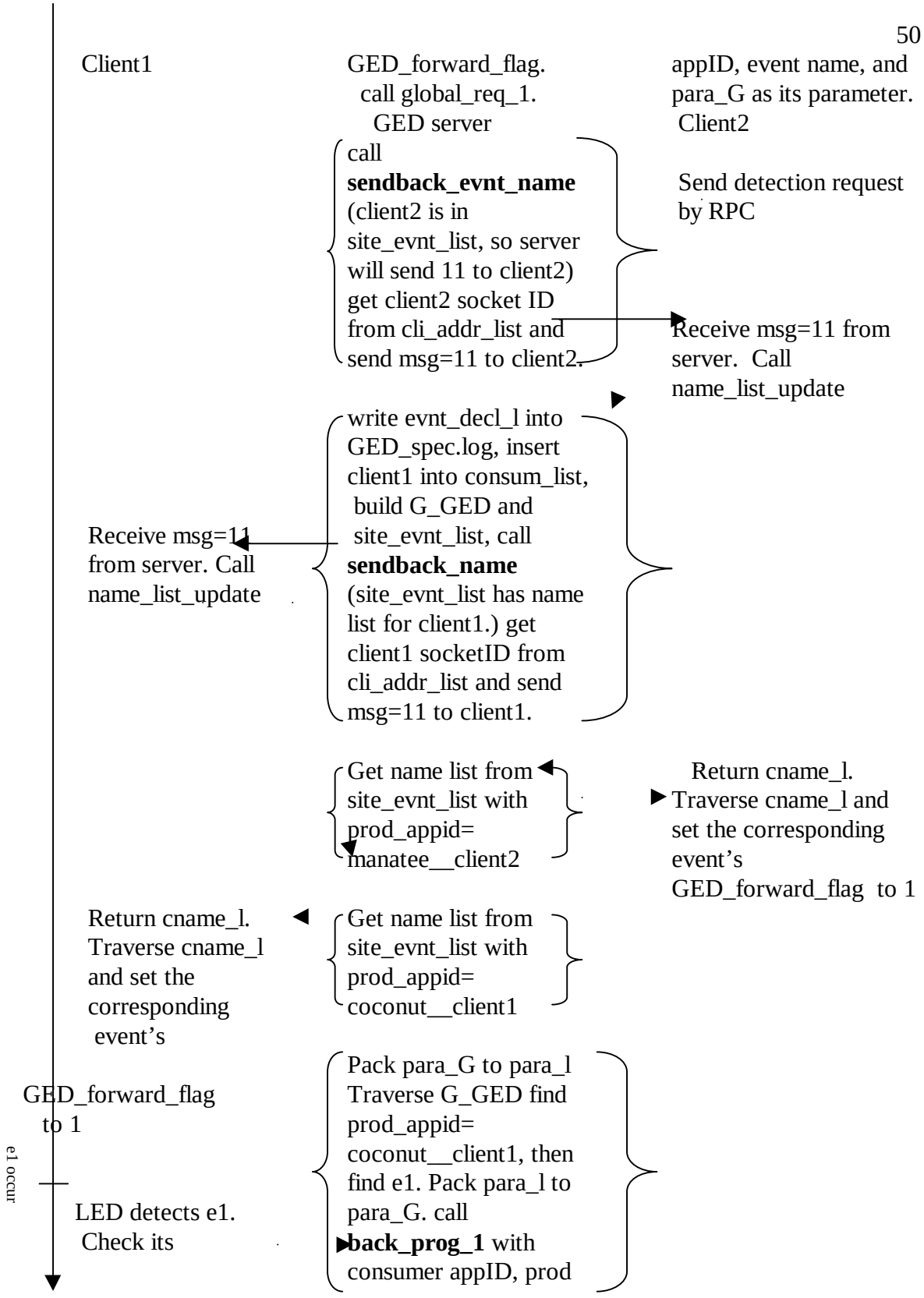
```
                append_times++;
        else if (a consumer has crashed)
              if (node has no event pending in the log file)
                 Insert the event to the buffer;
              else
                 Insert events to the buffer from log file until buffer is full or no
                 more events are pending in the log file;
        else if (a new consumer has came in)
              if (the number events in buffer < the new buffer size limit)
                 Insert the event to the buffer;
              else if (the number of events in buffer > the new buffer size limit)
                 Delete the exceeding events from the buffer and change the
                 blsn in the log file;
               else if (the number of events in buffer = the new buffer size limit)
                  append_times++;
    }
  else if (the consumer has considered crashed)
    {
      if (node's para_list != NULL)
         delete its para_list;
     }
```

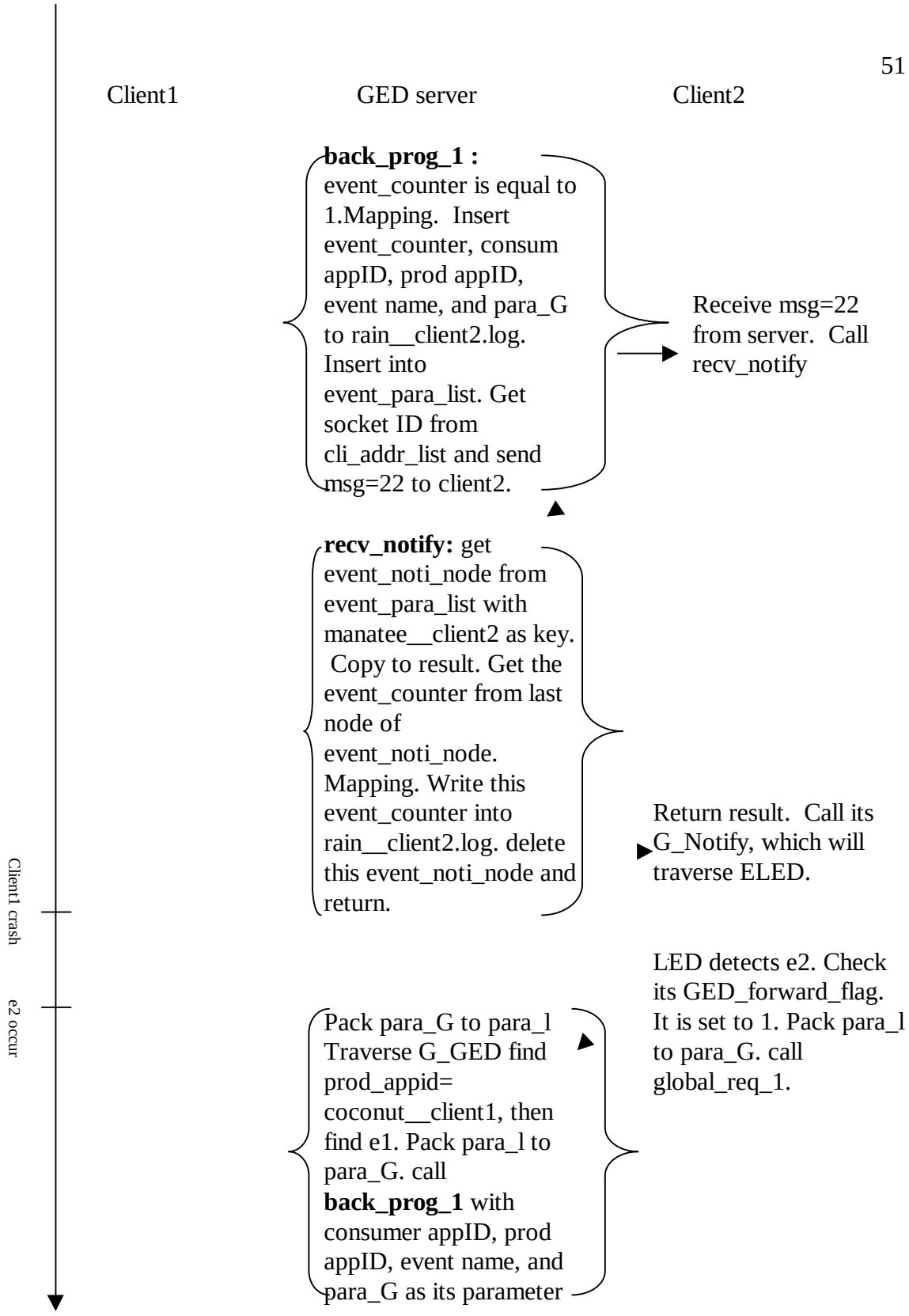
7.7 A Sample Scenario

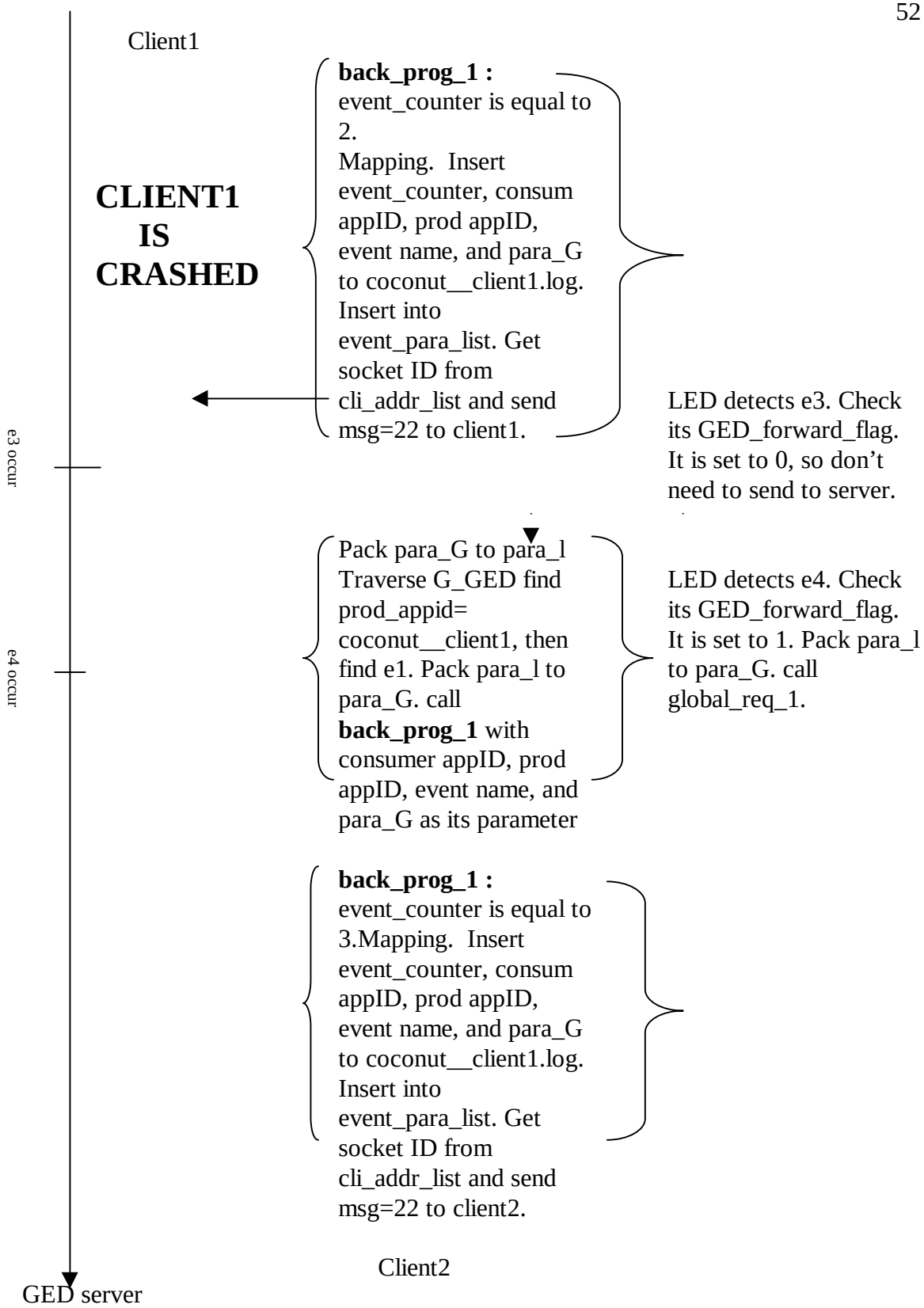
To summarize what we have been talked about in this chapter, let us go over an scenario and start at the very beginning. There are two applications listed below. Client1 is running on coconut and client2 is running on manatee. Below are the codes for application Client1 and Client2 and also a time chart which specify what has been done at each site at a given point in time.

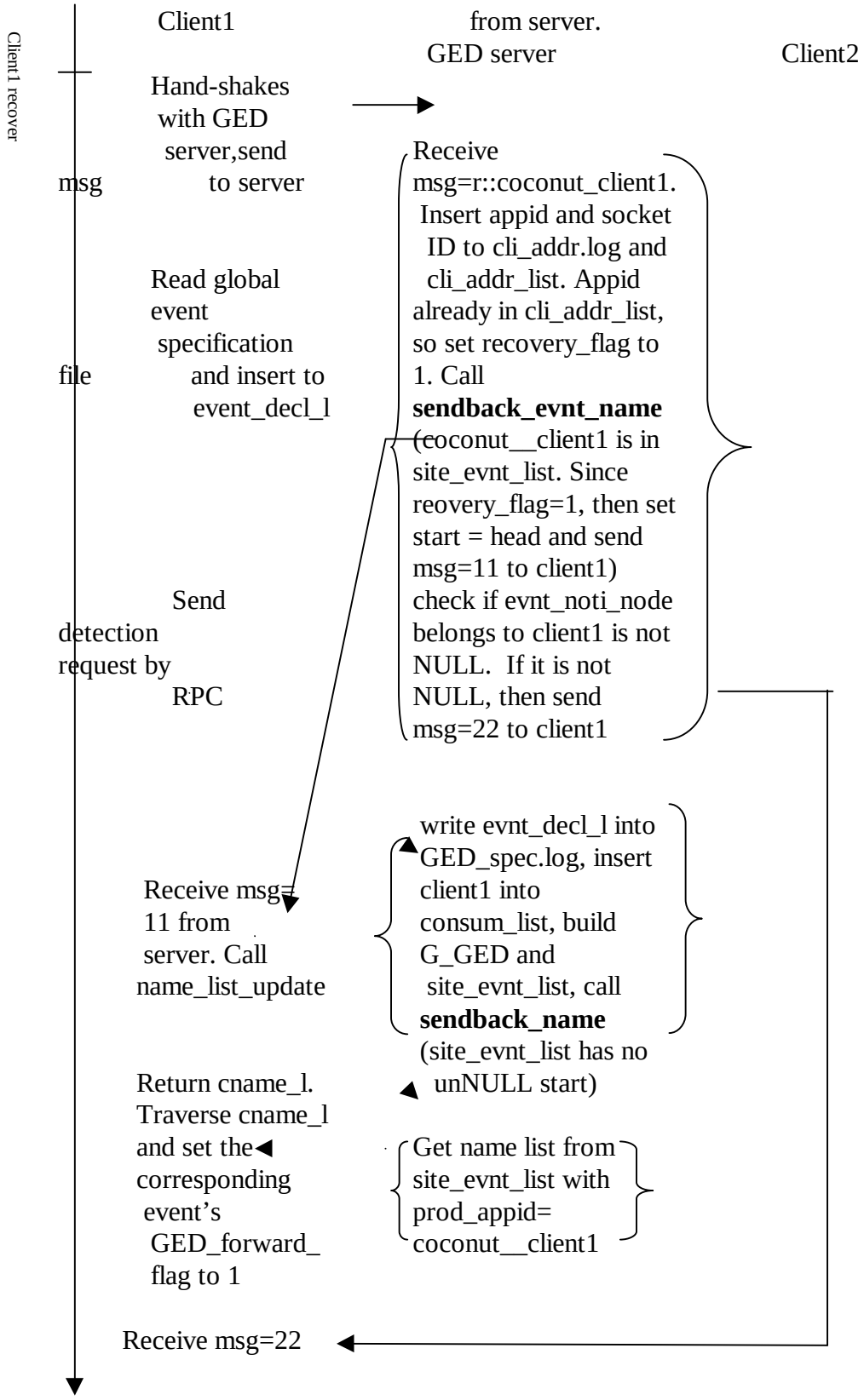
```
Client1                          Client2

event e1(end) void temp1         event e2(end) void temp2
G_e2 = e2::rain__client2         event e3(end) void temp3
G_e4 = e4::rain__client2         event e4(begin) void temp4
                                 G_e1 = e1::oconut__client1
```

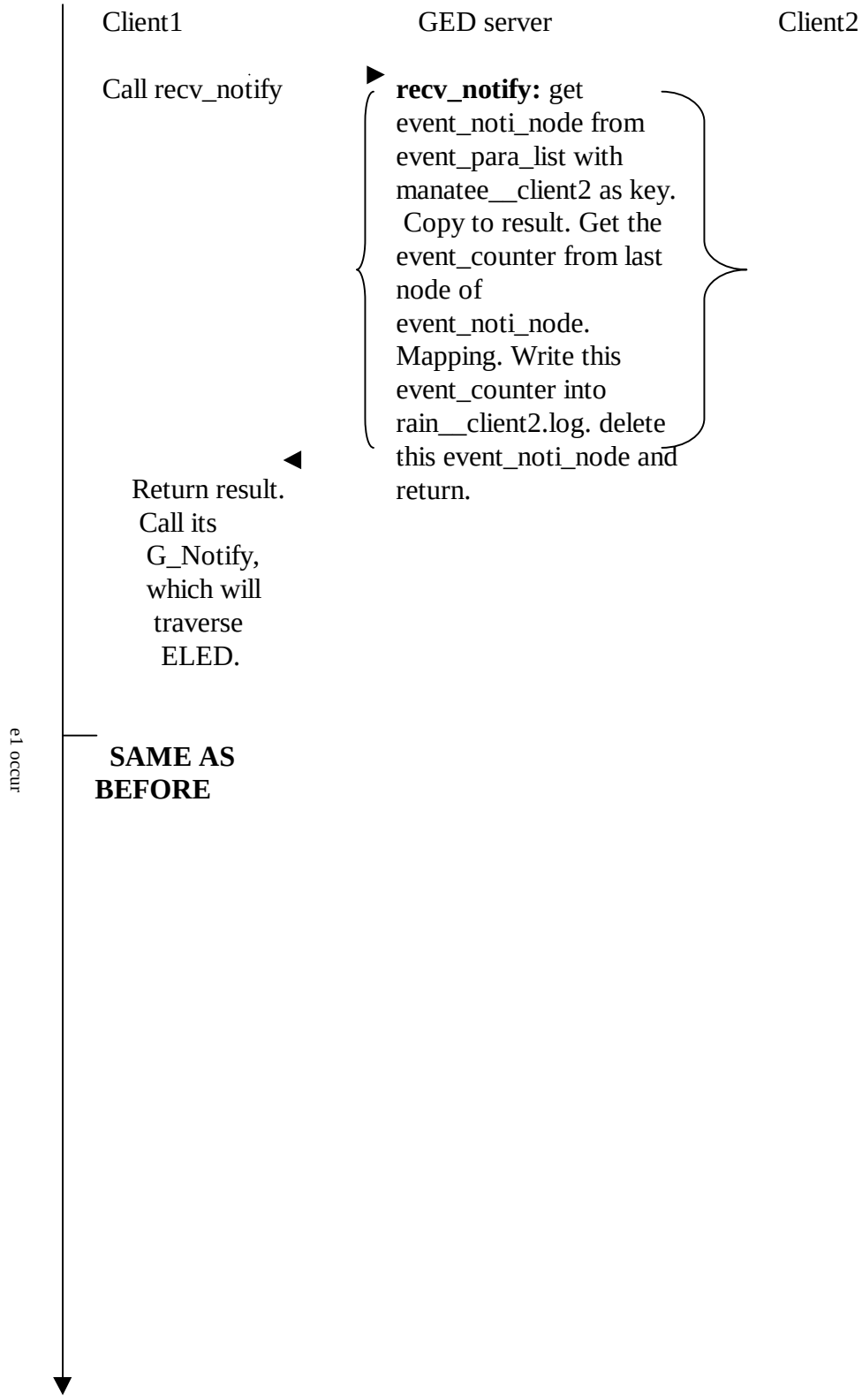

Start GED server
Client1
GED SERVER
Client2
Read configuration file, Create config_list. Check for recovery .
Start client1
Hand-shakes with GED server, send msg to server
Receive msg= i::coconut_client1 insert appid and socket ID to cli_addr.log and cli_addr_list. Call Sendback_evnt_name (site_evnt_list is NULL at this point.)
Read global event specification file and insert in event_decl_l
Send detection request by RPC
Call global_notify
write evnt_decl_l to GED_spec.log, insert client1 into consum_list, build G_GED and site_evnt_list, call sendback_name (site_evnt_list has name list for client2, but it is not connect yet. So nothing is sent.)
Start client2
Receive msg= i::manatee_client2 insert appid and socket ID to cli_addr.log and cli_addr_list.
Hand-shakes with GED server, send msg to server.
Read global event specification file and insert into event_decl_l


Client1
GED_forward_flag.
call global_req_1.
GED server
call
sendback_evnt_name
(client2 is in
site_evnt_list, so server
will send 11 to client2)
get client2 socket ID
from cli_addr_list and
send msg=11 to client2.
appID, event name, and
para_G as its parameter.
Client2
Send detection request
by RPC
Receive msg=11 from
server. Call
name_list_update
write evnt_decl_l into
GED_spec.log, insert
client1 into consum_list,
build G_GED and
site_evnt_list, call
sendback_name
(site_evnt_list has name
list for client1.) get
client1 socketID from
cli_addr_list and send
msg=11 to client1.
Receive msg=11
from server. Call
name_list_update
Get name list from
site_evnt_list with
prod_appid=
manatee__client2
Return cname_l.
Traverse cname_l and
set the corresponding
event's
GED_forward_flag to 1
Return cname_l.
Traverse cname_l
and set the
corresponding
event's
Get name list from
site_evnt_list with
prod_appid=
coconut__client1
GED_forward_flag
to 1
e1 occur
LED detects e1.
Check its
Pack para_G to para_l
Traverse G_GED find
prod_appid=
coconut__client1, then
find e1. Pack para_l to
para_G. call
back_prog_1 with
consumer appID, prod


Client1
GED server
Client2
back_prog_1 : event_counter is equal to 1.Mapping. Insert event_counter, consum appID, prod appID, event name, and para_G to rain__client2.log. Insert into event_para_list. Get socket ID from cli_addr_list and send msg=22 to client2.
Receive msg=22 from server. Call recv_notify
recv_notify: get event_noti_node from event_para_list with manatee__client2 as key. Copy to result. Get the event_counter from last node of event_noti_node. Mapping. Write this event_counter into rain__client2.log. delete this event_noti_node and return.
Return result. Call its G_Notify, which will traverse ELED.
Client1 crash
e2 occur
LED detects e2. Check its GED_forward_flag. It is set to 1. Pack para_l to para_G. call global_req_1.
Pack para_G to para_l Traverse G_GED find prod_appid= coconut__client1, then find e1. Pack para_l to para_G. call back_prog_1 with consumer appID, prod appID, event name, and para_G as its parameter


Client1
CLIENT1 IS CRASHED
back_prog_1 :
event_counter is equal to 2.
Mapping. Insert event_counter, consum appID, prod appID, event name, and para_G to coconut__client1.log. Insert into event_para_list. Get socket ID from cli_addr_list and send msg=22 to client1.
LED detects e3. Check its GED_forward_flag. It is set to 0, so don't need to send to server.
e3 occur
Pack para_G to para_l Traverse G_GED find prod_appid= coconut__client1, then find e1. Pack para_l to para_G. call back_prog_1 with consumer appID, prod appID, event name, and para_G as its parameter
LED detects e4. Check its GED_forward_flag. It is set to 1. Pack para_l to para_G. call global_req_1.
e4 occur
back_prog_1 :
event_counter is equal to 3.Mapping. Insert event_counter, consum appID, prod appID, event name, and para_G to coconut__client1.log. Insert into event_para_list. Get socket ID from cli_addr_list and send msg=22 to client2.
Client2
GED server


Client1 recover
Client1
from server.
GED server
Client2
Hand-shakes with GED server,send msg to server
Receive msg=r::coconut_client1. Insert appid and socket ID to cli_addr.log and cli_addr_list. Appid already in cli_addr_list, so set recovery_flag to 1. Call sendback_evnt_name (coconut__client1 is in site_evnt_list. Since reovery_flag=1, then set start = head and send msg=11 to client1) check if evnt_noti_node belongs to client1 is not NULL. If it is not NULL, then send msg=22 to client1
Read global event specification file and insert to event_decl_l
Send detection request by RPC
write evnt_decl_l into GED_spec.log, insert client1 into consum_list, build G_GED and site_evnt_list, call sendback_name (site_evnt_list has no unNULL start)
Receive msg= 11 from server. Call name_list_update
Return cname_l. Traverse cname_l and set the corresponding event's GED_forward_ flag to 1
Get name list from site_evnt_list with prod_appid= coconut__client1
Receive msg=22


Client1
GED server
Client2
Call recv_notify
recv_notify: get event_noti_node from event_para_list with manatee__client2 as key. Copy to result. Get the event_counter from last node of event_noti_node. Mapping. Write this event_counter into rain__client2.log. delete this event_noti_node and return.
Return result. Call its G_Notify, which will traverse ELED.
e1 occur
SAME AS BEFORE

CHAPTER 8
CONCLUSIONS AND FUTURE WORK

8.1 Conclusion

This thesis extends earlier work on Global Event Detector in Sentinel to make Global Event Detector robust. Global Event Detector was to monitor events in a distributed database environment. An event specification language SNOOP, its preprocessor spp, and a Local Event Detector were also developed as part of Sentinel to define and detect events in a centralized environment. Since Sentinel supports events across multiple applications, we need to address the robustness and recovery aspects of the system.

This thesis uses the existing architecture of Global Event Detector and adopted the Write Ahead Log, Log Sequence concept, and Locks to complete the goal of this thesis.

Chapter 1 and chapter 2 describe recent work on distributed database management systems and related work on recovery and consistency.

Chapter 3 provides a summary of SNOOP language and its preprocessor.

Chapter 4 gives an overview of event detectors. Architectures of LED and GED are discussed.

Chapter 5 provides a detail description on tables that are used for GED, including Extended Local Event Detector.

Chapter 6 gives the design and implementation of how to make GED robust. Several recovery alternative algorithms are presented and compared. Write Ahead Log is chosen and Log Sequence Number is adopted to incorporated with log files. Exclusive locks are introduced for consistence issue. A buffer management technique is also introduced to solve the problem when memory is full. The last section of this chapter gives a simple example of how everything incorporates with each other.

8.2 Future Work

- A check point mechanism can solve the problem of ever growing log files.
- Define rules at the GED through an interface. This can be used for propagating event notifications from one application to another. Updating data across databases can be realized in this approach.
- Use a distributed transparent mechanism such as CORBA for generalizing the concept proposed in this thesis.
- Have a multi-threaded GED server.
- Implement one event log file instead of multiple files.
- Use of operator P* (as well as A and A*) at the server to propagate information from one client to the other. This can be used for asynchronous transfer of data, update propagation etc

REFERENCES

[1] A. Schade An Event Framework for CORBA-Based Monitoring and Management System. IBM Research Division, Zurich Research Laboratory. Jan. 1997.

[2] S. Schwiderski. Monitoring the Behavior of Distributed Systems. Ph.D thesis, University of Cambridge, London, 1996.

[3] C. Mohan, Don Haderle, Bruce Lindsay, Hamid Pirahesh and Peter Schwarz. ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging. IBM Almaden Research Center and IBM Santa Teresa Laboratory.

[4] M. Stonebraker. The Design of the POSTGRES Storage System. EECS Department, University of California, Berkeley, CA.

[5] H. Korth, Abraham Silberschatz. Database System Concepts. McGraw Hill, Inc. University of Texas at Austin, 1991

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

[6] M. Ozsu and P.Valduriez. Principles of Distributed Database Systems. Prentice Hall.University of Alberta, Edmonton, Canada, 1991.

[8] H. Liao. Global Events in Sentinel: Design and Implementation of a Global Event Detector. Master thesis, University of Florida, Gainesville, 1997.

[9] D. Mishra. SNOOP: An Event Specification Language for Active Database. Master's thesis, University of Florida, Gainesville, 1991.

[10] S. Chakravarthy and D. Mishra. Snoop: An Expressive Event Specification Language for Active Databases. *Data and Knowledge Engineering*, 14(10):1-26, October 1994.

[11] W. Stevens. UNIX Network Programming. Prentice-Hall, Inc. Englewood Cliffs, New Jersey, 1990.

APPENDIX
LOG FILES

- Event Log File

65 65 27 coconut__consprod rain__prod CONSPROD_cs3 PRODUCE_e2 PRODUCE 13 881872860 11711 PRODUCE_e2 g 1 25 ! NULL# 0.000000 0 NULL# NULL# NULL#
34 coconut__consprod rain__prod CONSPROD_cs3 PRODUCE_e2 PRODUCE 15 881872890 10459 PRODUCE_e2 g 1 25 ! NULL# 0.000000 0 NULL# NULL# NULL#
40 coconut__consprod rain__prod CONSPROD_cs3 PRODUCE_e2 PRODUCE 17 881872920 963
1 PRODUCE_e2 g 1 25 ! NULL# 0.000000 0 NULL# NULL# NULL#
46 coconut__consprod rain__prod CONSPROD_cs3 PRODUCE_e2 PRODUCE 19 881872950 8639 PRODUCE_e2 g 1 25 ! NULL# 0.000000 0 NULL# NULL# NULL#
51 coconut__consprod rain__prod CONSPROD_cs3 PRODUCE_e2 PRODUCE 21 881872980 7756 PRODUCE_e2 g 1 25 ! NULL# 0.000000 0 NULL# NULL# NULL#
56 coconut__consprod rain__prod CONSPROD_cs3 PRODUCE_e2 PRODUCE 23 881873010 6858 PRODUCE_e2 g 1 25 ! NULL# 0.000000 0 NULL# NULL# NULL#
58 coconut__consprod rain__prod CONSPROD_cs3 PRODUCE_e2 PRODUCE 27 881873070 5106 PRODUCE_e2 g 1 25 ! NULL# 0.000000 0 NULL# NULL# NULL#
65 coconut__consprod rain__prod CONSPROD_cs3 PRODUCE_e2 PRODUCE 29 881873100 4243 PRODUCE_e2 g 1 25 ! NULL# 0.000000 0 NULL# NULL# NULL#

- GED_spec.log

  juice__cons 5503 0 CONSUM_c1 1 juice__cons global sugar__prod
  PRODUCE_e1 * !NULL 5503 0 CONSUM_c2 1 juice__cons global sugar__prod
  PRODUCE_e2 * !NULL 5503 0 CONSUM_c3 1 juice__cons global sugar__prod
  PRODUCE_e_AND * !NULL 5503 0 CONSUM_c4 1 juice__cons global
  coconut__consprod CONSPROD_cs1 * !NULL 5503 0 CONSUM_c5 1
  juice__cons global coconut__consprod CONSPROD_cs2 * NULL#
  coconut__consprod 25060 0 CONSPROD_cs3 1 coconut__consprod global
  sugar__prod PRODUCE_e2 * !NULL 25060 0 CONSPROD_cs4 1
  coconut__consprod global rain_prod PRODUCE_e_AND * NULL#
- client_addr.log

  juice__cons üãø0
  sugar__prod ÙÇãø%
  coconut__consprod Ñ:ãø&
  juice__cons ü2ãø0
  coconut__consprod ÑQãø&

BIOGRAPHICAL SKETCH

Jennifer Chun-Chun Sung was born on August 17, 1973, at Taipei, Taiwan. She received her Bachelor of Science degree in computer science from Old Dominion University, Norfolk, VA, in August 1995. In the spring of 1996, she started her graduate studies with a major in computer and information science and engineering at the University of Florida. She will receive her Master of Science degree in computer and information science and engineering from the University of Florida, Gainesville, in May 1998. Her research interests include active and object-oriented databases.