RECOVERABLE GLOBAL EVENT DETECTOR FOR DISTRIBUTED ACTIVE APPLICATIONS

The members of the Committee approve the masters thesis of Sreekant Thirunagari

Sharma Chakravarthy
Supervising Professor ____________________________________

Mohan Kumar ____________________________________

Alp Aslandogan ____________________________________

RECOVERABLE GLOBAL EVENT DETECTOR FOR DISTRIBUTED ACTIVE APPLICATIONS

by

SREEKANT THIRUNAGARI

PRESENTED TO THE FACULTY OF THE GRADUATE SCHOOL OF
THE UNIVERSITY OF TEXAS AT ARLINGTON IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS
FOR THE DEGREE OF

MASTER OF SCIENCE IN COMPUTER SCIENCE

THE UNIVERSITY OF TEXAS AT ARLINGTON
May 2002

To my parents

ACKNOWLEDGMENTS

I would like to thank my advisor, Dr. Sharma Chakravarthy, for his great guidance and support, and for giving me an opportunity to work on this project. I am also thankful to Dr. Mohan Kumar and Dr. Alp Aslandogan for serving on my committee.

I would like to thank Pratyush Mishra for maintaining a well-administered research environment and being so helpful at times of need. Thanks are due to Weera Tanpisuth and Raman Adaikkalavan for their help and fruitful discussions during the implementation of this work. I would like to thank all my friends at ITLAB. I also thank my friends Sirish Davuluri, Shashidhar Govind, Nishanth Vontela and Venugopal Cherukupalli for their support and encouragement.

I am thankful to my parents and brother for their constant support and encouragement throughout my academic career without which I would not have reached this position.

TABLE OF CONTENTS

page
ACKNOWLEDGMENTS iv

LIST OF TABLES ix

LIST OF FIGURES x

ABSTRACT 12

CHAPTER 1 14

INTRODUCTION 14

CHAPTER 2 17

RELATED WORK 17

2.1 Database Recovery 17
2.1.1 ARIES 18
2.1.2 Shadow Paging 20
2.2 JMS 21
2.3 Recovery of C++ GED 22
CHAPTER 3 23

Summary of Local and Global Event Detectors 23

3.1 Local Event Detector 23
3.1.1 Event Specification Interfaces and Usage 23
3.1.2 Event Graph and Propagation of events 25
3.1.3 An Overview of Components in Local Event Detector 26
3.2 Global Event Detector 27
3.2.1 Global Events 28
3.2.2 Architecture 29
3.2.3 Global Event Detection Site 30
3.2.4 Communication Module 31
3.2.5 Type of Messages 32
3.2.5.1 Detection Request Message 33
3.2.5.2 Event Notification Message 33
3.2.6 Global Event Graph 33
3.2.6.1 Dynamic Graph Construction: 34

CHAPTER 4 35

Design Issues for Persistence and Recovery 35

4.1 Requirements to Make GED Recoverable 35
4.2 Persistence 36
4.2.1 Notification Message Log File 37
4.2.2 Persisting other Data Structures 39
4.2.3 Persisting the Event Graph 41
4.3 Recovery 41
4.3.1 Server Recovery 42
4.3.2 Failure of Producers and Consumers 43
4.4 Buffer Management 44
4.4.1 Initial design 44
4.4.2 Alternate Design 47
4.4.3 Sending messages to consumers 48
4.4.4 Current Design 49
4.5 Buffer Manipulations 50
4.6 Guaranteed Delivery of Events 51
4.7 Extensions to Configuration File 52
4.7.1 Extensions to Global Configuration File 53
4.7.2 Extensions to Application Configuration File 54
4.8 LOCKS 55
4.8.1 Mutex 56
4.8.2 ReadWrite 56
4.8.3 Semaphore 56
4.9 Summary 57
CHAPTER 5 58

Implementation of Persistence 58

5.1 Implementation of Log Files 58
5.1.1 Basics of Java Object Serialization 58
5.1.2 Tailoring Serialization 61
5.1.3 Notification Message Log File 63
5.1.4 Other Log Files 65
5.1.5 Log Compression 65
5.2 Implementation of Buffer Management 66
5.2.1 Object Store 67
5.2.2 Notification Dispatch Thread 68
5.2.3 Pull Message Thread 69
5.3 Implementation of Recovery and Other Locks 70
5.3.1 Implementation of Recovery Lock 71
5.3.2 Other Locks 71
5.4 Configuration File 72
5.5 Summary 74
CHAPTER 6 75

Implementation of Recovery 75

6.1 GEDInterface 75
6.1.1 Client Address List (Hashtable clntAddrsHt) 76
6.1.2 Producer Event List (Hashtable prod_DectectnReqstHt) 77
6.1.3 ConsumerList (Hashtable glbEvntName_consumerList) 79
6.2 GlobalNodeManager 80
6.2.1 Hashtable glbEvntNm_GlbEvntNd 80
6.2.2 Hashtable glbEvntNm_GlbEvntHndle 81
6.3 BufferManager 82
6.3.1 Main Memory Buffers (objectStore1 and objectStore2) 82
6.3.2 Slow Consumer buffers (Hashtable clientId_logEvntCounter) 84
6.4 Processing Undelivered Messages 84
6.5 Summary 85
CHAPTER 7 86

Sample Scenario 86

CHAPTER 8 92

Conclusions and Future Work 92

8.1 Conclusion 92
8.2 Future Work 92
REFERENCES 94

BIOGRAPHICAL SKETCH 96

LIST OF TABLES

Table Page

Table 3-1 Common API’s used in Event Detectors 28

Table 3-2 APIs to Create Global Events 29

Table 3-3 Alternative Architectures 30

Table 5.1 Global Configuration File 73

Table 5.2 Application Configuration File 74

Table 7-1 Global Configuration File for this execution 86

LIST OF FIGURES

Figure Page

Figure 3-1 Local Event Graph 26

Figure 3-2**:** The communication layer between LED and GED 32

Figure 3-3: Global event graph for detecting global events 34

Figure 4-1 Notification Message Log File 38

Figure 4-2 Overview of GED Server 39

Figure 4-3 Buffer Manager 45

Figure 4-4 Log Header 46

Figure 4-5 Consumer Buffers 46

Figure 4-6 Object Store and Consumer Buffers 47

Figure 4-7 Refined Buffer Manager 50

Figure 4-8 Global Configuration File 54

Figure 4-9 Locks package 55

Figure 4-10 Lock usage 57

Figure 5-1 Code showing serialization process 60

Figure 5-2 Code showing Serialization into a ByteArrayStream 63

Figure 5-3 Index Table 64

Figure 5-4 Buffer manager 67

Figure 5-5 Slow Consumer Buffers 69

Figure 5-6 Recovery Lock 71

Figure 6-1 Producer Event List 78

Figure 6-2 Consumer List 79

Figure 6-3 Message Queue 83

ABSTRACT

RECOVERABLE GLOBAL EVENT DETECTOR FOR DISTRIBUTED ACTIVE APPLICATIONS

Publication No.________

Sreekant Thirunagari

The University of Texas at Arlington, 2002

Supervising Professor: Sharma Chakravarthy

Active applications support mechanisms that enable them to respond automatically to events that are taking place and are thus able to monitor and react to specific circumstances of relevance to an application. To support the reactive behavior a description mechanism called ECA (Event-Condition-Action) rules are used.

Based on ECA rule paradigm, Local Event Detector (LED) provides active capability to various kinds of applications enabling them to react to local events. The Global Event Detector (GED) is a server based on the notification/subscription model. It also uses the ECA rule paradigm in order to support active event monitoring capability in a distributed environment.

Distributed applications are prone to a variety of failures like client crashes, system failures and network failures. For reliable operation, any system in a distributed environment should be able to handle such failures. GED as any other application is prone to system crash and in addition, clients connected to GED can fail.

Earlier work on GED did not handle the robustness of GED to failures. The motivation for this thesis is to have reliable event detection and propagation by designing a recoverable GED that can be brought to previous consistent state following various types of failures, can continue to provide services when it recovers from failures and guarantees delivery of events. GED must be able to manage event buffers and accommodate the slow consumers.

This thesis provides buffer management, persistence and recovery capabilities to the GED server. All the information needed for recovery must be in stable storage at the time of recovery. Write Ahead Logging (WAL) concept is used to persist the appropriate information required to recover the GED from a crash. Buffer manager module manages the main memory used to store incoming events and handles buffer over flows and, the required read/write access to secondary storage.

CHAPTER 1

INTRODUCTION

Active applications support mechanisms that enable them to respond automatically to events that are taking place and are thus able to monitor and react to specific circumstances of relevance to an application. To support the reactive behavior, a declarative mechanism called ECA (Event-Condition-Action) rules [1] are used. These rules have three components: an event, a condition, and an action. An *event* is an instantaneous, atomic occurrence of interest at a specific point in time to which the rule may be able to respond. The *condition* part of the rule evaluates the condition using the context in which the event has taken place. The *action* describes the task to be carried out by the rule if the relevant event has taken place and the condition has evaluated to true. When an event occurs the rule is triggered. If the condition associated with the rule evaluates to true, action is executed.

Based on ECA rule paradigm, Local Event Detector (LED) [2] provides active capability to various kinds of applications including relational database systems. It uses flexible and expressive event semantics provided by SNOOP [3] [4]. LED is well suited for monitoring complex changes within an application. LED allows the applications to define ECA rules on local events. To extend the event and rule specification capabilities of applications, from events occurring in their local address space to events occurring in different address spaces, a Global Event Detector has been developed.

The Global Event Detector (GED) [5] is a server based on the notification/subscription model. It also uses the ECA rule paradigm in order to support active event monitoring capability in a distributed environment. It enables an application to monitor an event or a combination of events occurring in multiple applications distributed over a network. Applications subscribe for a remote events and the LED, transparently, sends a detection request messages to GED. GED, in turn, notifies the clients who produce these events to start sending them to GED. GED, then, notifies the subscribers as and when it receives a notification of event of interest from the producer. It is also possible for the clients to request the GED to detect composite events and notify when those composite events occur. GED detects composite events of interest based on event detection requests and event notifications it gets from the client application.

Distributed applications are prone to a variety of failures, such as client crashes, server failure and network failures. For reliable operation, any system in a distributed environment should be able to handle such failures. GED, as any other application, is prone to system crash and in addition, clients connected to GED can fail.

Current implementation of the Global Event Detector does not address the robustness to system failures. All the event information is kept in main memory and sent to clients. If memory is not sufficient in GED, events can be lost. If GED crashes all the event information along with global event graph is lost. To recover from a crash all the information needed should be available in stable storage at the time of recovery. With out event persistence and recovery, GED and all client applications need to restart. When a consumer is not responding or slow, the producer will still send events; these events are lost due to the lack of in memory buffers.

In order to have a reliable event detection and propagation, a recoverable Global Event Detector that can be brought to previous consistent state following various types of failures, can tolerate client failures, and can continue to provide services when it recovers from failures is needed. GED must be able to manage event buffers and accommodate the slow consumers.

This thesis provides buffer management and recovery capabilities to the GED server. GED is enabled with an option to choose the persistent mode of operation. All the information needed for recovery must be in stable storage at the time of recovery. In persistent mode Write Ahead Logging (WAL) concept is used to log the appropriate information required to recover the GED from a crash. Buffer manager module takes care of event buffers in case of buffer overflows and, handles the required read/write access to secondary storage.

The outline of this thesis is as follows: Chapter 2 reviews the work related to different ways of providing persistence and recoverable capability to an application. Chapter 3 summarizes the architecture and usage of existing local and global event detector systems. Chapter 4 explains the design issues associated with the buffer manager and providing event persistence and recovery capabilities to GED. Chapter 5 goes into the implementation details of buffer manager and data persistence onto stable storage. Chapter 6 explains the logging and recovery of different data structures required for the GED recovery. Chapter 7 shows an example scenario demonstrating the robustness of GED to system failures and client crashes. Chapter 8 concludes the thesis and discusses the future work.

CHAPTER 2

RELATED WORK

This chapter reviews the work related to providing persistence and recoverable capabilities. It discusses the persistence and recovery related to databases. It discusses the database approach, recovery with ARIES and explains the similarities and difference of GED recovery approach compared to ARIES algorithm. It also explains the shadow paging mechanism and its disadvantages. It then discusses the features of JMS that can be useful for making GED recoverable. Finally, it discusses C++ version of robust Global Event Detector.

2.1 Database Recovery

DBMS ensures the atomicity and durability of its transactions to provide fast recovery. Atomicity of transaction implies that all the actions in it are executed or none. Durability implies that all the effects of a successful transaction are persist even after system crash. A DBMS has a recovery manager that maintains relevant information in normal execution of transactions in order to enable it to perform its task in the event of a system crash. A log of all the modifications to the database is saved on stable storage. It ensures that the log entries describing a change to database are written to stable storage before the change is made. The log enables recovery manager undo the actions of aborted and incomplete transactions and redo the actions of committed transactions. If no-force approach is used, in case of crash, some transactions updates might be still in the buffer pool. Such changes must be identified and written to disk. The changes made by transactions that did not commit prior to crash might have been written to disk because of steal approach. Such changes must be identified and using the log and then undone. Recovery manager ensures atomicity by undoing the actions of uncommitted transactions and ensures durability by making sure that all actions of committed transactions are persistent.

Initial approach to database recovery was an UNDO/REDO approach. Later on, ARIES and other variants of write ahead logging based recovery mechanisms replaced this. Shadow paging is one other way to provide database recoverability.

2.1.1 ARIES

Algorithm for Recovery and Isolation using Event Semantics (ARIES) [6] [7] is a Write Ahead Logging (WAL) based recovery mechanism. This is a database approach and is an improvement over conventional undo/redo approaches prior to ARIES. It supports fine granularity locking and partial rollbacks. Aries uses log files to record the actions that cause changes to recoverable data objects. It records all transaction into a log. The log is considered as an ever-growing sequential file. This log is critical for ensuring a transaction's committed actions are reflected in the database despite various types of failures and that its uncommitted actions are undone. Log files are stored on stable storage, which is non-volatile, remains intact and available across system failures. Aries supports page-oriented redo and logical undo, thus achieving efficiency and high concurrency.

Information logging can be of two types, physical and operational. In physical logging, before update and after update values of specific fields within the object are stored. In operational logging, the operations that were performed on the object are recorded. Operation logging permits the use of high concurrency lock modes, which exploit the semantics of the operations performed on objects.

The WAL protocol asserts that the log records representing changes to some data must already be on stable storage before the changed data is allowed to replace the previous version of that data on nonvolatile storage. The system is not allowed to write an updated page to the nonvolatile storage version of the database until at least the undo portions of the log records, which describe the updates to the page, have been written to stable storage.

To enable the enforcement of this protocol, systems using the WAL method of recovery store in every page the LSN of the log record that describes the most recent update performed on that page. LSN is a unique log sequence number assigned to the record when that record is appended to the log. LSNs are assigned in ascending sequence.

Recovery process in ARIES is divided into three phases analysis, redo and undo phases. During recovery, the first thing it does is analysis, which is to repeat history. In redo phase, history is repeated to 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. In undo pass, 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. Basic features of ARIES can be summarized as follows:

- Simplicity
- Operational and value logging
- Partial rollbacks
- Multi-granularity recovery
- Page-oriented recovery
- Logical undo
- Red0.Undo only as necessary
- Don't redo something that is already done
- Flexible storage management
- Flexible buffer management
- Minimal overhead

To provide the event persistence and recovery capabilities to the existing GED we have adopted Write Ahead Logging mechanism. ARIES algorithm is not used in complete, because there is no UNDO phase in GED recovery.

DBMS maintains certain information in normal operation in order to enable it to recover in the event of crash. With ARIES, DBMS uses WAL to persist the log records indicating the changes made by a transaction so that they can be redone in case of a crash. Aries is a steal, no-force approach. In case of a crash, all the committed transactions are repeated thus bringing the database to its state prior to crash. All the actions of uncommitted transactions are undone, thus restoring the database to previous consistent state.

Similar to DBMS, in normal operation, GED stores certain information to restore its state in case of a crash. Unlike DBMS, GED has non-transactional approach. This introduces a window of failure. If a crash occurs during the processing of event massage from client the state of the GED is lost. WAL is used to persist the event messages GED receives from clients. All the events that are received are persisted before they are processed. This avoids the loss of unprocessed events in the event of crash and aids in buffer management. Any update to the GED state during the event detection process is persisted by writing information that reflects this change to log before updating the in memory data values.

ARIES algorithm uses LSN as log record id that enables it to fetch the record with one disk access. GED uses a similar concept. Each event message that comes on to GED is assigned a unique id called Event Sequence Number. GED uses this id to keep track of the message stay on GED. Given a message ESN the message can be read from the log in single file access. ESN is also used in buffer management and recovery of main memory event buffers in case of crash.

As there are no transactions and transaction atomicity, recovery of GED involves only the redo portion. The history is repeated by reading the log files and restoring the state of GED prior to crash. There is no undo in GED recovery.

2.1.2 Shadow Paging

It is based on maintaining a dual mapping between pages and their location on disk. One mapping represents the current state of a segment being modified; the other represents a previous backup state. At any time, the backup state can be replaced by the current state without any data merging. The basic idea in shadow paging is that existing data is never overwritten, but instead modified pages are always written to new locations on disk, and a mapping is used to keep track of the current location of each page. The mapping is called the page table. Shadow paging [8] [9] implies force policy, that is, all modified data must be written to non-volatile storage before a transaction can commit.

Before a transaction is committed, current page table is written to disk. Current page table is made the shadow page table by overwriting the disk address of shadow page table by the address of current page table.

Shadow paging is better compared to log-based systems in cases where fast recovery can be extremely important and applications with large read-only transactions mixed with small updates. Advantages of shadow page mechanism over log-based mechanism are that the overhead of log-record output is eliminated and recovery from crashes is much faster (no undo or redo). Disadvantages of shadow page mechanism over log-based mechanism are:

1. Every transaction commit need to write actual data blocks, current page table and its disk address to stable storage.
2. Garbage collection imposes additional overhead and complexity
3. Offers no help for fine granularity locking for concurrent transactions and hence difficult to be adapted for concurrent transactions.

2.2 JMS

Message Oriented Middleware (MOM) products allow separate business components to be combined into a reliable, yet flexible, system. JMS [10] provides a common way for Java programs to create, send, receive and read a MOM system's messages.

JMS is a set of interfaces and associated semantics that define how a JMS client accesses the facilities of an enterprise-messaging product. JMS incorporates persistence in message delivery mode. JMS supports two modes of message delivery, PERSISTENT and NON-PERSISTENT.

In NON_PERSISTENT mode, message is not logged to stable storage. A JMS provider must deliver a NON_PERSISTENT messag*e at-most-once*. This means that in case of JMS provider failure, it may lose the message, but it must not deliver it twice. In PERSISTENT mode, extra care is taken to insure that the message is not lost in transit due to a JMS provider failure. A JMS provider must deliver a PERSISTENT message *once-and-only-onc*e. This means a JMS provider failure must not cause it to be lost, and it must not deliver it twice.

JMQ can be used for guaranteed delivery of event messages from server to client and vise versa. The main objective of this thesis is to provide a recoverable GED server. This implies that JMQ can provide only the persistence with respect to the messages exchanged. Persistence of GED state and recovery from crash should be handled separately. JMQ itself comes with a lot of additional functionality and communication overhead [5]. The functionality, persistent message delivery, provided by it is not worth paying the overhead associated with it.

JMS specifications [10] say, “The use of PERSISTENT messages does not guarantee that all messages are always delivered to every eligible consumer. ”

2.3 Recovery of C++ GED

C++ version of GED [11] [12] is robust to sever and client failures. It follows log based recovery approach. The algorithm used here similar to ARIES. Write Ahead Logging is used to ensure that all the changes are recorded on stable storage. It uses LSN to keep track of the message objects on the secondary storage and for guaranteed delivery of events to clients. With the aid of LSN, it keeps track of number messages sent and number of messages to be sent. It uses individual buffers for each client to store the notification messages of events subscribed by this client. Each consumer buffer has a separate log file that persists all the messages received for this client. This log file aids in recovery of buffers in the event of crash and buffer management of the client buffers in normal operation.

Unlike the Java GED, which constructs the event graph dynamically, C++ GED constructs event graph statically. C++ clients use Snoop preprocessor. A Snoop Preprocessor transforms the event and rule definitions written in Snoop to the conventional C++ programming code. It generates the global event specification file, which contains the information used by the server for detecting the event defined outside of a local application. Based on this information, the global event detector constructs the event graph for the detection of primitive and composite global events. Hence, it does not face the problem of persisting the event graph. In case of crash, the event graph is reconstructed from the event specification file. Only the information that is propagated through the graph is persisted. The state of the graph is restored using this information. For the same reason it can support client recovery, whereas Java GED cannot. This will be explained in detail in chapter 4.

CHAPTER 3

Summary of Local and Global Event Detectors

3.1 Local Event Detector

This section summarizes the Java Local Event Detector (LED)[2] and its functionality. Based on ECA rule paradigm, local event detector supports active capability to various kinds of monitoring applications including relational database systems. It uses flexible and expressive event semantics provided by SNOOP[3] [4] [13]. The LED has been used to develop an agent that works with various commercial Relational DBMSs (such as Oracle[14], DB2[15] and Sybase[16]). The local event detector is well suited for monitoring complex changes within an application.

3.1.1 Event Specification Interfaces and Usage

Local event detector provides active capability to an application by detecting the occurrence of local primitive and local composite events defined as part of the application. The application interacts with the local event detector through a set of interfaces shown in Table 3-1. Following are the steps for using the event detector.

1. Application has to initialize an agent by invoking the *initializeECAAgent ()* or *initializeECAAgent (String agentName*) method.

When initialized for the first time, the system initiates a new default ECA agent. If it is not for the first time, it will return the existing defaultECAAgent to the application. Multiple ECAAgents identified with different names can be initialized using *initializeECAAgent (String agentName)* API. Each agent is responsible for monitoring events defined in that agent and performing appropriate actions.

2. After initialization, the application can start defining events (both primitive and composite), and rules.
    a. Define primitive events using *createPrimitiveEvent* API.
    b. Define composite events using *createCompositeEvent* API.
    c. Define rules using *createRule* API.

Using the ECAAgent initialized in step 1, the application can define a primitive event by invoking the *createPrimitiveEvent* method. Application has to supply name for the primitive event, name of the class in which the method associated with the event is defined, the event modifier (begin or end), and the complete method signature as parameters to this method. This method returns the event handle associated with the primitive event, which can be used for defining the composite events with this event as a constituent, storing the parameter of the event, and signaling the method invocation of event to the detector.

The application can define a composite event in a specific ECAAgent by invoking *createCompositeEvent* method on that ECAAgent. To define a composite event, the application specifies the operator type and event handles of constituent events obtained from previous declarations. These constituent events can be either primitive events or other composite events. Based on the number of constituent events, SNOOP composite event operators can be classified into two categories, binary and ternary. Two basic API's that can be used to define binary SNOOP operators, such as AND, OR, and SEQUENCE with two constituent events and ternary SNOOP operators, such as NOT, PERODIC, and APERIORDIC are shown in Table 3-1.

To define a rule associated with an event the application should provide the handle corresponding to that event, a condition method name, and an action method name. The condition and action are defined as methods in the associated class. Basic API used for defining a rule is shown in Table 3-1.

3. Raise an event.
    a. Insert parameters of different data types using *insert ()* APIs provided by ECAAgent.
    b. Raise the event.

The application can insert any primitive data type or object through *insert ()* APIs shown in Table 3-1. In the method defined as primitive event, arguments of condition and action methods need to be inserted into the parameter list using the event handle before raising the event. These parameters can be used in evaluating the condition and performing actions.

An event can be raised at the beginning or at the end of the method corresponding to that event. The applications are provided with two APIs, *raiseBeginEvent* and *raiseEndEvent* for this purpose. These are also shown in Table 3-1.

LED detects the occurrence of the events through the invocation of *raiseBeginEvent* or *raiseEndEvent* methods. All the associated rules are triggered when the event occurs. The condition is evaluated. If the condition is satisfied, the action will be performed.

3.1.2 Event Graph and Propagation of events

LED constructs an event graph when events are declared and uses that graph during event detection. Each node in the event graph is either a primitive event or a composite event defined in the application. Primitive events become the leaf nodes. Constituent events of a composite event can be primitive or other composite events. Hence, non-leaf nodes in the graph represent composite events. Each event node contains a list of rules associated with it and a list of composite events subscribed for its occurrence. LED supports instance level rules on primitive events as there is a single object associated with a primitive event. Instance level composite events do not make sense, as there is one object associated with each constituent occurrence. The primitive event node contains an instance-based multiple rule list and an event subscriber list, while the composite event node contains only one rule subscriber list and an event subscriber list. On the occurrence of the event, its corresponding node is updated and the information propagated to the subscribed intermediate node (composite events). The occurrence is propagated by means of an event table called PCTable.

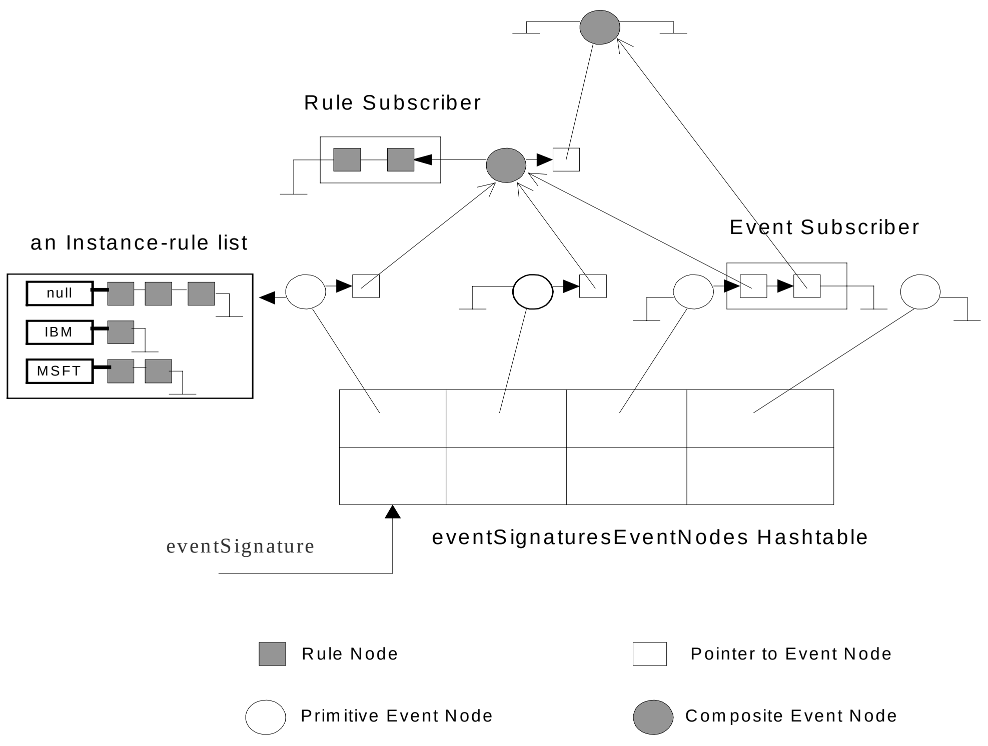


Figure 3-1 Local Event Graph

3.1.3 An Overview of Components in Local Event Detector

The building blocks of local event detector are events, rules, ECAAgent, rule scheduler, and event detector thread. Every event defined in the application has an event handle. EventHandle object stores method signature associated with the event, class name in which the event is defined, and a list of parameter lists associated with that event. Event handle has been introduced to encapsulate several pieces of information pertaining to an event and to reduce the amount of information the user needs to keep track of.

A rule consists of a condition and an action. These are specified in *classname.methodname* format. The conditions and actions are implemented as methods in a Java class. LED uses Java reflection to refer to the classes in which the condition and action methods are implemented. Whenever the rule is triggered, the *RuleThread* object is instantiated and is inserted into the rule queue if rule scheduler is turned on. Rule scheduler later executes it.

The ECAAgent provides interfaces to define events and rules, insert parameters, and signals an event occurrence to the event detector. The ECAAgent class maintains data structures to refer to event nodes with their names and method signatures associated with them. Application thread is separated from the event detector and works in conjunction with an event detector thread (*LEDThread*) through a buffer (*NotifyBuffer*).

The event detector thread is responsible for detecting events and firing rules. Whenever a primitive event occurs, all the relevant information about its occurrence is wrapped into an object called *NotifyObject*. This *NotifyObject* is put into the buffer, and is processed by the *LEDThread*. Running in an infinite loop, the event detector thread keeps fetching *NotifyObjects*, notifying occurrences of events, propagating the parameters to the internal nodes of the event graph, and firing the associated rules.

3.2 Global Event Detector

The local event detector is well suited for monitoring complex changes within an application. The capabilities of LED are limited to a single address space. To extend this event detection capability to distributed environment Global Event Detector is designed.

The Global Event Detector (GED)[5] is a server based on the notification/subscription model. It uses the ECA (Event-Condition-Action) rule paradigm in order to support active event monitoring capability in a distributed environment. It detects composite events of interest based on event detection requests and event notifications it gets from the client application. This section summarizes the global event detector.

Create Primitive Event API	CreatePrimitiveEvent (String eventName, String className, EventModifier modifier, String methodSignature)
Create Composite Event API	createCompositeEvent (EventType operator, String eventName EventHandle ehOne, EventHandle ehTwo) createCompositeEvent (EventType operator, String eventName, EventHandle ehOne, EventHandle ehTwo, EventHandle ehThree)
Create Rule API	createRule (String ruleName, EventHandle eh, String ruleName, String condMethod, String actionMethod) createRule (Object targetInstance, String ruleName, EventHandle eh, String ruleName, String condMethod, String actionMethod)
Insert Parameter API	insert (EventHandle [] eventHandleArray, String varName, long longValue) insert (EventHandle [] eventHandleArray, String varName, float floatVal) insert (EventHandle [] eventHandleArray, String varName, Object object)
Raise Event API	raiseBeginEvent (EventHandle [] eventHandleArray, Object instance) raiseEndEvent (EventHandle [] eventHandleArray, Object instance)

Table 3-1 Common API's used in Event Detectors

3.2.1 Global Events

Like LED, GED also uses SNOOP for the flexible and expressive event semantics. GED uses the event names to refer to event nodes. Hence, to make the event names unique, global event name is composed of event name on the detection site, name of the application that is detecting this event, and host name on which this application is running. The event detection and network communication details are transparent to users. The events on GED are mainly classified into two types, global primitive events and global composite events.

Create Global Primitive Event API	CreatePrimitiveEvent (String consEventName, String className, String prodEventName, String appName, String machName)
Create Global Composite Event API	createCompositeEvent (EventType operator, String eventName, EventHandle ehOne, EventHandle ehTwo) createCompositeEvent (EventType operator, String eventName, EventHandle ehOne, EventHandle ehTwo, EventHandle ehThree)

Table 3-2 APIs to Create Global Events

Global primitive event is an event that is defined and detected outside of current or local application. On the site of detection, this could be a primitive or composite event. The API used to define global primitive event is shown in Table 3-2.

Global composite event is an event that is composed by event operators and at least one of its constituent events is a global event. The global composite event specification is identical to the local composite event specification. The internal mechanism will determine the type of composite event (local or global) and the site of global composite event detection at run-time. The details of composite event detection site are described later in this section. The APIs used for defining a global composite event are shown in Table 3-2.

3.2.2 Architecture

GED uses client/server architecture. Table 3-3 summarizes the other architecture alternatives considered for GED. This approach introduces a global event detector as a server. Each application communicates only with the server and hence, it does not have to know the identities of other applications. Server is responsible for managing the subscription/notification aspect of global event detection. Determining what global events to detect and where to detect, the server allows clients (applications) to share information. The global event detector partially relieves the applications of event detection. Each application still contains the local event detection module. Applications are loosely coupled.

A consumer (of events detected in another application) can subscribe to remote events through the global event detector. Running in the background on the server, the global event detector keeps track of subscriptions on the server site and forwards the request to the appropriate application that generated the event (a producer). Once the event occurs in the producer site, the global event detector is notified which forwards the notification to the consumer. The maximum number of messages between clients and server is: 2*[X+Y] where X is used to represent a number of consumer applications and Y is either 2 or 3 (binary or ternary operator).

Client/server architecture with remote procedure call	Client/server architecture with	Client/server architecture with object request broker (ORB)	
		CORBA	JAVA-RMI
Characteristics - Encapsulates details of the network interfaces - Most RPCs are blocking communication **Drawbacks** - Not support object communication. - Allow only primitive data type.	**Characteristics** - Provides message router and message queues for message passing between applications - Proprietary software from venders **Drawbacks** - Not all MOM implementations support all operating systems and protocols. - Can't modify internal infrastructure to accomplish our goals	**Characteristics** - To manage interaction between clients and servers via ORB - Supports primitive data types, and a wide range of data structures, as parameters **Drawbacks** - Not support the transfer of objects, or code. - No garbage collection (CORBA)	**Characteristics** - Provides the mechanism by which the server and the client communicate and pass information (objects) back and forth **Advantages** - Portable across many platforms - Support object communication across network - Cost ***This approach is used to implement the Global Event Detector***

Table 3-3 Alternative Architectures

3.2.3 Global Event Detection Site

In a distributed setting, data is exchanged among applications. The communication strategy is a key factor to reduce the communication cost and it is critical to system performance. The cost of communication is described as:

Communication Cost = Frequency * (Overhead + Occupancy)

The overhead is the time to initiate the transfer. The occupancy is defined as the time it takes for transmitting the data. Frequency is the number of times a message is sent. The factors affecting the system performance, size and number of messages exchanged among applications, should be minimized to obtain a better performance. In the global event detector, there are two approaches for detecting global composite events.

First approach can be termed the mediator approach. In this approach, GED acts merely as a mediator passing event occurrences as messages among the clients. It forwards the notification messages it receives from producers to consumers. Event graph of the global composite event is constructed at the local site. All the event detection is done at the local sites.

In the second approach, the global composite events are detected at the server site as well. The GED server not only receives and forwards the event information, but also detects any composite event of interest. As the composite events are detected on server site, GED does not forward every occurrence of the constituent primitive event to the consumer of the composite event. Rather, the event graph is constructed on the server site and GED sends the notification back to the corresponding client application only when the composite global event is detected.

The location where the global composite events are detected has a major impact on the number of messages passed between the server and client applications. Detecting all the composite events at local sites is costly when all the constituent events are global. Consequently, the second approach seems better in this situation. However, when at least one of the constituent events is a local event, first approach has less message passing.

A global composite event can be detected either at the local site or at the server site. The site where a global composite event is detected is determined by its constituent events at run time. The global composite event is detected at the GED when all of its constituent events are the global events; whereas, it is detected at the local site when one of its constituent events is a local event.

3.2.4 Communication Module

The earlier implementation of Local Event Detector [ref…] is only aware of the events defined in its address space. It cannot send the event detection request to another application. Therefore, a communication interface is introduced to handle the remote calls so that the existing local event detector can exchange information with other applications.

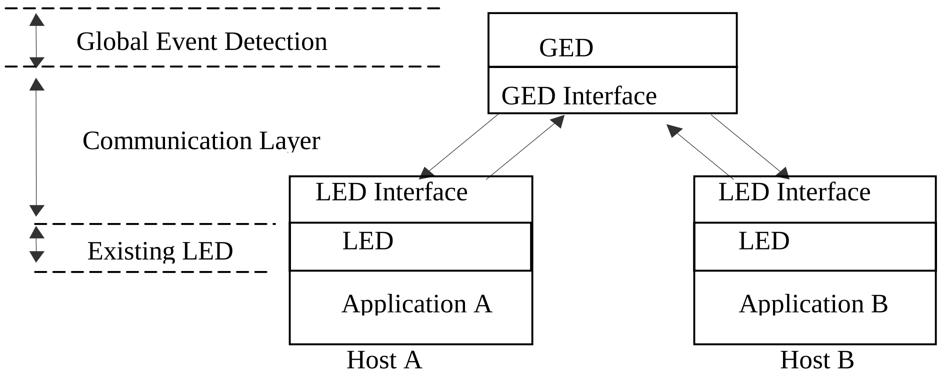


Figure 3-2**:** The communication layer between LED and GED

The LED interface is introduced to facilitate the communication between LED and GED. It is responsible for looking up the remote objects and making remote invocations to send an event detection request or event notification to the server. It also has a listener that waits for incoming messages such as event notification and event detection request from the server.

On the server site, the GED interface is designed not only to forward the messages from one application to another application, but also to store some data to construct the global event graph and additional information for global event detection. Every application needs to register with the server through this interface. The communication layer between GED and LED is shown in Figure 3-2.

3.2.5 Type of Messages

Typically, there are two types of messages passed between clients and server. One is, the client application communicates with the server to request the detection of a global event, and receives event notification from the server when that event occurs. Second is, the client receives an event detection request from the server and sends it the notification when that event is detected at its site. Hence, the messages exchanged between clients and the GED server can be classified into two types, *eventNotificationMessage* and *detectionRequestMessage*.

3.2.5.1 Detection Request Message

The *detectionRequestMessage* is packed with event name, application name, and host name of the global event and context bit information to capture the useful semantics of the application. Once the GED server receives the message, it creates the global event node representing the event on the server site. The *sendBackFlag* is set in order to specify that there is at least one application waiting for the occurrence of this event. Then the server forwards the message to the application that defines the event. The producer application unpacks the package and sets the *forwardFlag* associated with event node to be true so that it will signal the event occurrence to the server.

3.2.5.2 Event Notification Message

The *eventNotificationMessage* contains parameter information about the global event. It contains an event table described earlier in section 3.1.2. Typically, the relevant information is recorded in the parameter lists in associated PCTable when the primitive event occurs. The occurrence of this primitive event is propagated to the internal node if it is a constituent event of the composite event. In addition, the parameter lists are inserted into the event table of internal node. Whenever the *forwardFlag* of the notified node is set, the *eventNotificationMessage*, including the PCTable, will be sent to the GED server.

3.2.6 Global Event Graph

The event graph is used for detecting composite events. Each event that is defined in the application is represented as an event node in the graph. The relationship among composite events and their constituent events forms an event graph. In the event graph for LED, each node has a list of event subscribers and a list of rule subscribers. When a composite event subscribes to its constituent nodes, the reference of composite event will be stored into the list of event subscribers. Similarly, when a rule is defined, it is stored in the list of rule subscribers. As mentioned earlier, the composite global events can be detected either on the local site or on the server.

The composite global event can also be detected on the GED server by using the event graph as shown in Figure 3-3. The leaf nodes of the global event graph represent the global primitive events. The internal nodes represent global composite events. Unlike the node of a local event graph, the global event node does not have a list of rule subscribers, since all the rules are locally executed at the application site. The rules that are defined on a global event are applied to the *Remote* event node on the application site instead. Each global event node contains only a list of event subscribers, which contains references to the global composite event nodes. In addition, each node maintains occurrences of events and their parameter lists in the *PCTable*. Whenever a global event occurs, it will check the *sendBackFlag* to check whether to send the event notification message to appropriate clients (or consumers).

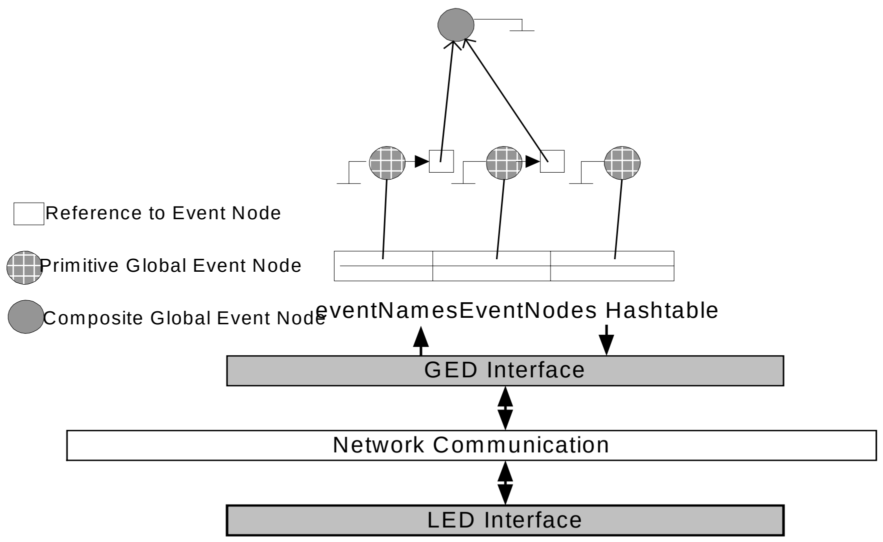

Reference to Event Node
Primitive Global Event Node
Composite Global Event Node
eventNamesEventNodes Hashtable
GED Interface
Network Communication
LED Interface

Figure 3-3: Global event graph for detecting global events

3.2.6.1 Dynamic Graph Construction:

The main emphasis of the design is to do things only on demand and avoid sending messages over the network unless it is necessary. Only on request from the consumers, the global event nodes are constructed at the GED server and *sendback* flag is set so that the occurrence of these events is notified to the consumers. Producers send notification messages of only those events for which server forwarded the detectionRequestMessages from consumers. This design ensures that the event nodes are constructed on demand and notification messages are passed over only when there are consumers for that event.

CHAPTER 4

Design Issues for Persistence and Recovery

Distributed applications are prone to a variety of failures like client crashes, system failures and network failures. For reliable operation, any system in a distributed environment should be able to handle such failures. GED as any other application is prone to system failures.

The existing system [5] summarized in previous chapter is a main memory system. All the event information and the state of the GED are stored in main memory. It assumes the availability of infinite main memory to handle the incoming event notification messages (explained in section 3.2.5.2). In case of a system failure, all the information in the volatile memory is lost. GED looses its state information along with the event graph resulting in a fresh start of GED and thus all its client applications. This chapter discusses the design details of persistence, recovery of GED and buffer management

4.1 Requirements to Make GED Recoverable

Following a crash, the system should be able to recover to its previous stable state and continue its normal operation there after. To recover from crash all the data needed to restore the state of GED should be made available even after crash. This can be achieved by persisting the information in log files.

The recovery process should be able to read the data from log files and reestablish the state of the GED prior to crash. It is desirable that, the overhead associated with the file IO during normal operation of GED and during its recovery is minimized.

All the incoming event notification messages are stored in a buffer and GED processes them from this buffer and dispatches them to corresponding event subscribers. As GED is prone to failures, loss of the unprocessed messages must be avoided.

As the resources could be limited, GED needs a buffer manager that can manage the main memory used to store the incoming messages from clients. This module should be able to handle the buffer overflows and required read/write to secondary storage. It should aid in recovery of main memory buffers. It should also take care of clients that cannot consume the events fast enough to keep pace with the GED.

The delivery of events and event detection should not be affected by the GED crash. Even when the GED is down, client applications continue to generate new events and send notification messages to GED. If the clients persist those events and resend them to GED after its recovery, event detection will be unaffected.

4.2 Persistence

Data persistence is the ability to keep data or information around even after a program ends. To recover a server from crash, enough information of the server must be persisted on stable storage so that the state of the server at the time of failure can be reconstructed at a later time. To handle recovery of GED server from failures, its state should be persisted. The key here is to store the state of GED sufficient to reconstruct it. As we cannot predict the crash, Write Ahead Logging (WAL) concept is used to store the state to stable storage. WAL ensures that before updating the data values in memory, appropriate information reflecting this update is persisted onto stable storage.

Along with the state of GED, the incoming notification messages should also be persisted. If the GED crashes before processing the event notification messages it received, they will be lost. Hence, to handle the recovery of unprocessed messages, all the incoming messages should be logged before they are processed. In case of a main memory buffer overflow, the incoming messages from the clients cannot be added to buffers and will be lost. If these messages are made available in stable storage and GED can write the unprocessed messages when there is empty space available in the buffer, it can avoid the loss of these messages. This is achieved by persisting the messages into a log file and reading them back later. They are persisted in Notification message log file, which is discussed in next section.

Java object serialization mechanism [17] is adopted to persist the data values. Java object serialization is a simple and flexible way to persist java objects based on copying of objects to and from streams. An object can be serialized by implementing the *Serialization* interface provided by Java and using the methods provided by the interface. It gives the ability to easily read and write entire objects and primitive data types, without converting to/from raw bytes or parsing text/ASCII data. Serialization is the mechanism to flatten the objects into a stream of bytes that can be written to disk or transferred over a network. De-Serialization is the mechanism to reconstruct the object from its serialized byte stream.

4.2.1 Notification Message Log File

This log file stores all the notification messages (event occurrences) arriving at the server. Each notification message is assigned a sequence number called Event Sequence Number (ESN) when it arrives at the server. Along with the notification messages, this log file also stores some additional information needed by the Buffer Manager in the process of normal buffer management and to recover the main memory buffers in case of a crash. It store three variables, buffESN -- the largest ESN in buffer, dESN -- the largest ESN that is dispatched to its consumers, pendingEvts -- counter that indicates the number of messages in the log to be pulled into main memory buffer, and itSize -- size of the Index Table of the log file in bytes.

In a traditional log file if an application wants to read an object stored in it, it has to do a sequential read of the file until it finds that object. This has an overhead of reading the unnecessary data from the file. This overhead grows as the file size grows.

To avoid the sequential reading of the log files, this design introduces a mechanism to index into the file and retrieve only the object that is required. All the message objects in log file are in a serialized byte stream form. To retrieve a specific object from this would require the exact position where the byte stream of this object would start and the length of the byte stream i.e., the number of bytes corresponding to this object. This information is collected at the time of writing the serialized object stream into the file. As mentioned before each message object is identified with a unique event sequence number. Byte size of the object and its offset in the file are stored along with the ESN in Index Table.

To read a specific object from the file, the index table is queried with ESN of the object to get its position and the number of bytes to read. The byte stream so read is de-serialized to obtain Java object.

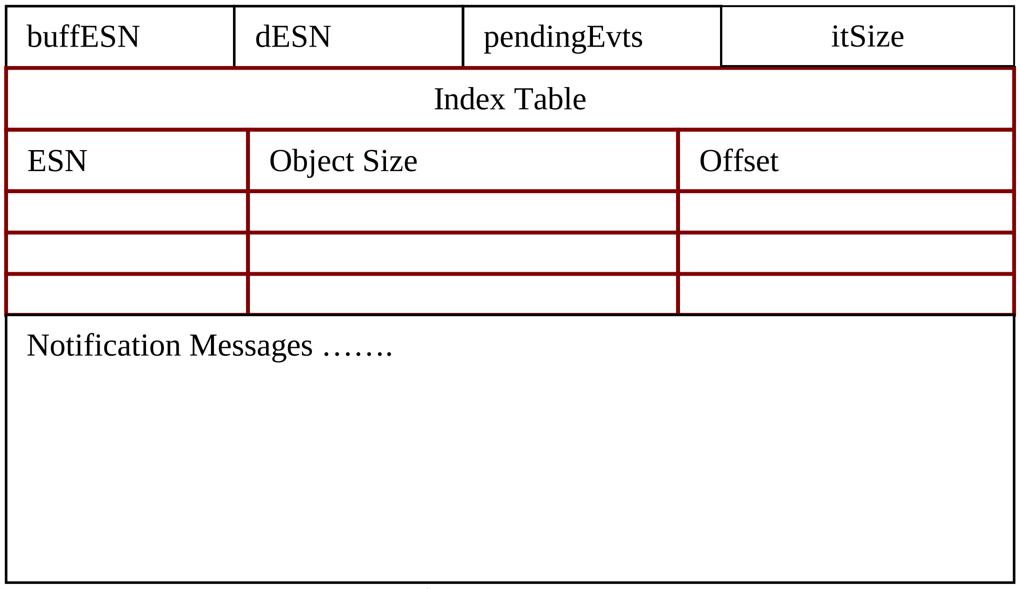


Figure 4-1 Notification Message Log File

This indexing capability comes at the cost of limited log file size. Unlike the ever growing traditional log files, this log file has a limited size specified in terms of number of notification messages. As GED has to log the notifications coming at runtime, it has to add new index records to the index table at runtime. Therefore, it needs to know exactly where in the log file the byte stream of this message should be written i.e., the offset of the object in file, and this position cannot change once it is written. This imposes the condition that all the objects that come before this notification message must be of fixed size. Nevertheless, the index table is populated only at runtime hence it grows at runtime. If the size of the index table were not limited, it would violate the condition mentioned above.

This problem can be avoided by two ways. One is to write the index table in a separate file so that the size of the index table can grow dynamically without effecting the offsets in the notification message log file. The other is to keep the index table in the same file and limit it to a fixed size. Index table is accessed each time a message is written to the file and read from the file. Hence, in the first approach, each write or read would result in opening and closing of two files. In second case, there is only one file open and close associated with each write or read. Nevertheless, the number of messages that can be held in the log would be limited and the log file needs to be compressed occasionally to avoid log operation failures. We assume that the overhead associated with opening and closing two files for each write or read operation is more than the overhead of compressing the log file occasionally.

This design limits the size of index table to a fixed number of bytes thus limiting the size of the file. The size of index table is mentioned in terms of number of message indexes it can hold. This value is read from the configuration file. GED creates the index table of fixed size, populated with the default values at the time of initialization. These default values are replaced with the actual values at runtime, thus maintaining the constant size.

Index table size is only a limit on the number of messages that can be stored in log file. This should not limit the number of messages GED server can handle. If the log file were full, the message logging would fail. To avoid the log overflow due to this limitation, a mechanism to compress the notification log file is provided. At any point of time, the dESN indicates the maximum ESN that was dispatched to its consumers. Ensured that all the messages with the ESN lower than dESN are already dispatched they do not need to be kept in the log file. These message objects are purged from the file and the index table is updated to reflect the current locations of the notification messages.

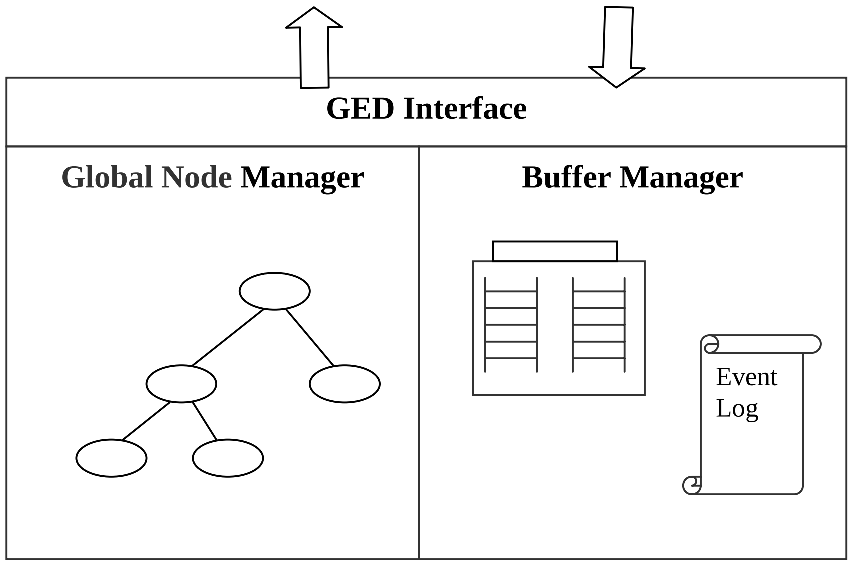


Figure 4-2 Overview of GED Server

4.2.2 Persisting other Data Structures

All the data structures that are a part of the GED's current state need to be persisted to enable GED recovery. As mentioned earlier, the key is to persist only that information which is required to restore the GED state. Each data structure is persisted onto a separate file. All these files are written in an append mode. Whenever the state of the data structure changes, instead of recording the entire data structure into the log, only the information that represents this change is written at end of the file. Reading these information bits step by step from the file, the data structure can be restored to its previous state.

As shown in Figure 4-2 GED server can be mainly divided into three parts. GED server can be mainly divided into three parts. The communication layer between the clients and the server becomes the first part. Processing the incoming event detection request messages and event notification messages to construct the global event graph and doing the global event detection becomes the second part. Managing the main memory buffer space available to store the notification messages before processing them and dispatching them to their corresponding consumers becomes the third part. Persisting GED state mainly involves persisting these three parts.

Not all the information in these objects is persisted. The threads involved in different data manipulations are tied to resources that are specific to this session of the virtual machine. It does not make any sense to serialize the thread for later use. Hence, they are re-instantiated in the recovery process and associated with the appropriate data values. The main memory buffers used to store incoming messages are not persisted. The notification message log file, explained in previous section, contains all the messages that come onto GED server. This log file can be used to extract the appropriate messages from the log and reconstruct the main memory buffers. Similarly, the remote interfaces used to communicate with the client applications are also not serialized. These remote interfaces are bound in the RMI registry with a standard name so that the client applications can look up for them in registry and use them later to communicate with GED. Once the system crashes, these objects will be no more available. Similar to threads, they are specific to this session of virtual machine and must be rebound at the time of recovery.

GED does a logical logging. Any information that can be restored or can be computed from already persisted data is not persisted. Depending on the semantics of the data structure, minimum amount of information required to restore its state is persisted.

The process of persisting the data structures and restoring them later in recovery process is explained in detail in chapter 6.

4.2.3 Persisting the Event Graph

Event graph cannot be persisted as any other data structure. Event graph is constructed dynamically. The event nodes are constructed only when there is a request from a client. GED needs to update it accordingly in stable storage along with the main memory copy.

When a new node is added to the existing event graph (a new event to the table), if it is a global composite event that consists of global primitive events it updates some data corresponding to these constituent events in memory and GED wants this to be reflected on secondary storage. Depending on the level at which the event is being added, number of internal nodes and leaf nodes that are modified varies. This could be as small as two primitive event nodes or as large as the entire event graph.

This may result in logging all the event nodes that are updated and keeping track of the sequence of these updates to re-do them during recovery process. This complicates the logging and recovery process.

At this point of time, we want to keep our recovery plan as simple as possible so we just write the updated in-memory version of hash table into a file. This introduces a window of failure. If the GED crashes while updating the event graph, the updates made to the event graph are lost because the stable storage only has the information that reflects the state of the graph before this update was started.

4.3 Recovery

There is a fundamental difference between a server recovery and an application recovery. Server recovery involves the restoration of the previous state of the server and it should resume to provide normal services after recovery. An application recovery along with the state restoration would involve pinpointing the point of crash and resuming from that point after recovery. When an application is started in initialize or resume mode it would always start executing from the first command. In resume mode, the application should be able to skip to the point of crash and then start executing the rest of the commands. A DBMS supports the application recovery [18] by relying on transaction commits to pinpoint the point of crash. It would thus know the operations to be redone and the operations to be undone. In case of applications where there is no notion of transactions, resuming an application is easier said than done. It involves an arduous deal of bookkeeping and cross checking for each statement executed.

GED is a server that supports the event detection in distributed environment by processing the incoming messages and sending out appropriate notifications. It provides services to clients that enable them to subscribe for remote events and receive notifications on the occurrence of these events. The expressive events semantics provided by Snoop enable it to detect the composite events composed of events occurring in different applications distributed over a network. It builds an event graph based on the event notification and event detection request messages it receives from the clients and uses it in the process of event detection. The recovery of GED server from a failure involves restoration of its state prior to crash and continue to provide normal event detection services.

We assume that system failures do not corrupt information on stable storage and the effect of failure is not spread beyond the point of crash. As mentioned in the previous section, the event graph persistence introduces a window of failure in which the loss of information is inevitable. WAL concept used in persisting all other information ensures that all the changes are recorded on to the stable storage and are available during the recovery.

4.3.1 Server Recovery

To recover from the crash, all the information required should be in stable storage at the time of recovery. The user is given a choice to run the GED in PERSIST mode. Event persistence capability provided in this design makes the GED server robust to failures. Write Ahead Logging (WAL) concept adopted in this design ensures that the information in stable storage reflects the GED state prior to crash. Apart from logging the notification messages arriving onto the server, all the data structures required to restore the system are logged to reflect their stable state prior to crash. When the server crashes, these log files are used to restore the GED state.

As shown in Figure 4-2 GED server can be mainly divided into three parts. The communication layer called GED Interface, Global Node Manager that maintains the global event graph and does the global event detection and the Buffer Manager that manages the main memory buffer space available to store the notification messages before dispatching them to their corresponding consumers.

Recovery of GED mainly involves recovery of these three parts. As a part of recovery GED, registers its communication interfaces with the RMI registry and instantiates it threads associated with message processing before the recovery is complete. This opens the GED for incoming messages from client applications, but the recovery process is not yet complete and GED state is not reestablished until the recovery is finished. Processing any messages would involve modifying the incompletely recovered GED state. Hence, the processing of any messages received during recovery should be deferred until the recovery is complete. Locking the entire recovery and performing it as an atomic operation achieve this. Access to all the data structures is restricted to the main thread until the recovery is complete.

The recovery process first recovers the state of the GED from log files. In the process, it opens the communication interface to the clients by rebinding the remote interfaces used to communicate with clients, and finally it takes care of any messages that were generated when GED was down. GED then continues normal operation. The recovery of GED server is explained in detail in chapter 6.

4.3.2 Failure of Producers and Consumers

Akin to GED, producers and consumers are also prone to errors. In the C++ version of the GED [11] [12], the event graph is built statically. The graph information is read from a specification file and it is used in the event detection process. The global event graph pertaining to a client is sent to the GED server at the time of initial handshake. GED uses this information and updates its event graph accordingly. When a consumer recovers from a crash, it can be started in INIT or RESUME mode. If it is started in RESUME mode, the previous spec file it sent will be retained and used for further communication with the client. If it is started in INIT mode, the client supplies a new spec file to the GED server.

In the Java version, the event graph is built dynamically. The main emphasis of the Java design is to do things only on demand and avoid unnecessary message passing over the network. Only on request from the consumers, the global event nodes are constructed on the GED server and any notifications of this event are sent back to the consumer. Producers send notification messages of only those events for which server forwarded the detection request messages from consumers. This design ensures that the event nodes are constructed on demand. There is no event graph specification file. Consumer detection requests are RMI calls to GED server. Unlike the C++ version where in all the event and rule definitions are collected into a spec file and sent to the server at the registration time, requests are sent to the GED server whenever it occurs in the client program execution. This makes the client recovery a program recovery. Java version doesn’t provide any mechanism to recover the client sessions.

To recover a client from crash we need to pinpoint the command on which the client application failed, recover the state up to that point and resume from that command. As creation of events is done dynamically, create event and create rule calls can come at any point in client code. When restarted, the application starts from the beginning. As the point at which client has crashed is not known, we cannot find how many client calls to LED/GED are executed and how many are pending. Hence, if a client crashes, current implementation cannot restore its state to the previous stable state. This would result in the loss of local event graph built during the previous run. Client has to start a fresh.

4.4 Buffer Management

The objective of the buffer manager is to provide a simple mechanism that takes care of the main memory constraints, handle required read and write on secondary storage. It should aid in recovery of the main memory buffers in case of system crash.

An abstract view of buffer manager would be as shown in the Figure 4-3. It should handle the main memory buffers used to store the event notification messages. It controls the addition of new messages to the buffers, dispatching of these messages to corresponding consumers. It should also handle the buffer overflows and use the secondary storage appropriately to avoid the loss of the messages in such cases.

4.4.1 Initial design

Each consumer has a buffer and a log file assigned to it. The size of consumer buffer is determined by the ratio of total available buffer space and number of clients. The consumer log file contains all the event notification messages (explained in section 3.2.5.2) for that consumer. Along with these messages, log file also contains header information, which is used in buffer management and recovery of the consumer buffers. To keep track of the messages on the server event sequence number (ESN) is used. ESN is a unique, monotonically increasing number assigned to messages on their arrival at the server. The log header shown in Figure 4-4 contains three values buffESN - the largest ESN in buffer, dESN - the largest ESN that is dispatched to the consumer, pendingEvts - counter that indicates the number of messages in the log to be pulled into main memory buffer.

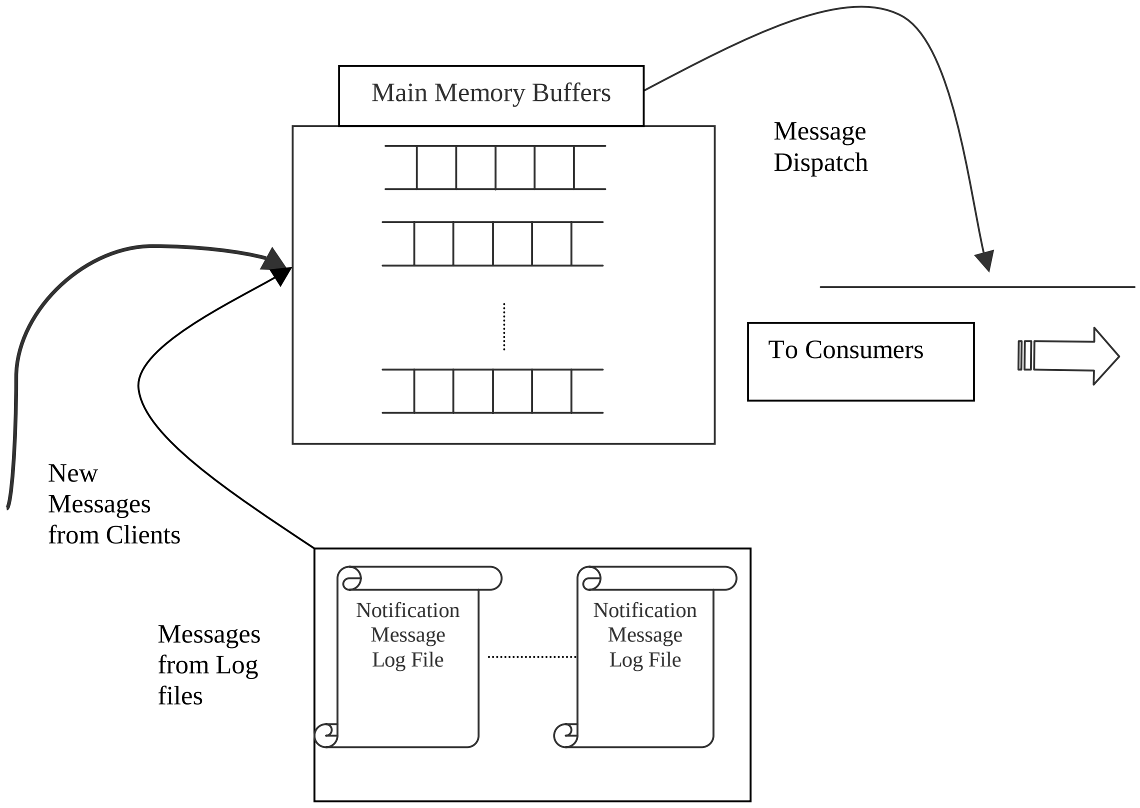

Main Memory Buffers
Message Dispatch
To Consumers
New Messages from Clients
Messages from Log files
Notification Message Log File
Notification Message Log File

Figure 4-3 Buffer Manager

Every message is logged before it is actually added to main memory buffer. If it cannot be added to main memory buffer, the necessary header information indicating that there is a logged message that needs to be pulled into main memory is stored in log. At a later point of time if there are any empty slots created by the dispatched messages, these logged messages will be pulled to main memory and are dispatched accordingly. At any point of time, the log file stores all the messages along with the information needed to recover the main memory buffer. Variables buffESN and dESN in log header indicate that at a given point main memory buffer contains messages with ESN between dESN and buffESN. This information is used to recover the main memory buffers in case of a crash. Data structure used for consumer buffers is shown in Figure 4-5. A hashtable was used for the purpose, where key is clientID and value is an object containing vector of messages along with rmi url (rmi://consIP/consID) for this client.

buffESN	dESN	pendingEvts

Figure 4-4 Log Header

The buffer manipulations are divided into two threads. First, there is a thread that handles the message dispatch from main memory buffers to consumers. Second thread pulls the logged messages, if any, from consumer logs into main memory buffers. In addition, there are RMI threads resulting from the client notification calls to GED, which add the newly arriving messages.

This design makes add and delete of messages to a particular consumer buffer independent of other consumer buffers. Having separate buffer and log files for each consumer enables us to concurrently handle the message delivery and buffer management. However, this concurrency comes at the cost of duplication and multiple file operations. Multiple consumers can subscribe an event. Hence, notification message for that event must be distributed to all its consumers. As this design has separate main memory buffer and a log files for each consumer, any message for multiple consumers must be stored in buffers of all these consumers and each consumer log has to be updated accordingly. For example in the Figure 4-5, message M1 is intended for all the three consumers as a result all the three consumer buffers and log files have a copy of M1. Duplication on secondary storage results in multiple file operations, which incur high performance cost.

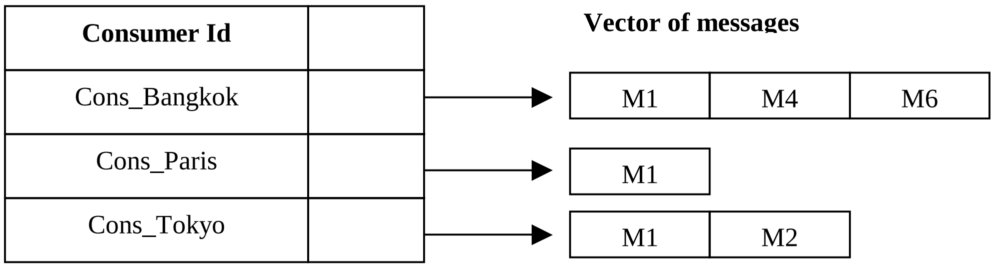


Figure 4-5 Consumer Buffers

4.4.2 Alternate Design

To avoid duplication of messages in main memory, we came up with an alternate design. In this design, all the messages arriving at the server are stored at a single place called object store. Instead of message itself, a reference to it is stored in all its consumer buffers. Figure 4-6 shows the object store and the consumer buffer. Still, this design uses multiple logs to enable independent writes and reads on secondary storage.

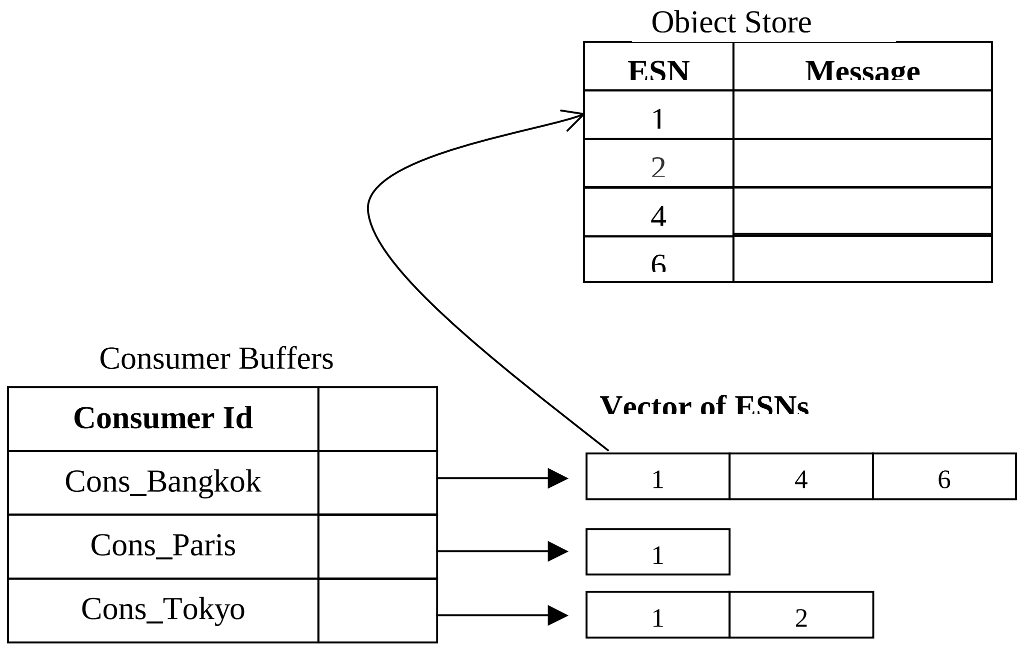


Figure 4-6 Object Store and Consumer Buffers

Object store is a data structure that stores all the messages according to the event sequence number associated with them. Message dispatch thread needs to extract messages with specific ESNs from object store. Hence, object store needs to store the mapping between the message and associated ESN to reduce the search time. Object store is implemented using a Java hashtable. In Java, a hashtable is structurally synchronized. This means, "*If multiple threads access this map concurrently, and at least one of the threads tries to modify the map structurally, it is synchronized internally*". (A structural modification is any operation that adds or deletes one or more mappings; merely changing the value associated with a key that an instance already contains is not a structural modification.) Thus any new add operations or delete operations on this table are synchronized, avoiding us from attaining any amount of parallelism with thread, which read messages and deletes them appropriately from main memory buffer and the thread, which adds the messages to main memory buffer.

Consider a case where a message for multiple (say n) consumers cannot be put into Object store due to the buffer overflow. It will be logged into all its consumer logs and each of these consumer logs reflect the information that they have message logged in their log file. This would result in ‘n’ file accesses. Pulling of the messages from log to main memory is done on consumer bases, i.e., a thread goes to each consumer log, checks for pending messages and pulls them to main memory. In this case, if a message is pulled to main memory, the application has to update all consumer logs that this message is in object store otherwise the thread makes multiple attempts to pull the same message from multiple logs if the corresponding info is not updated in all the logs. In either case, application has to do more ‘n’ file accesses. This sums to a total of 2n file accesses for a message with n consumers.

4.4.3 Sending messages to consumers

In both the above-mentioned designs, when a message arrives on the server, following is the order of processing:

1. Find all consumers for this message
2. Log the message in each consumer log
3. If it can be added to main memory (Consumer buffer/Object store), it is added (In later design, the reference is kept in each consumer buffer) and the header info (buffESN) is updated in each consumer log accordingly.
4. If it cannot be added to object store, the header info (pendingEvts) is updated in each consumer log accordingly.

The event semantics used by GED depend heavily on the time and order of occurrence of the events. Hence, while dispatching the event notification messages that arrive onto the server the global order of the messages should be maintained. The dispatch thread dispatches the messages on consumer basis. This thread goes over all the consumers’ buffers sequentially and dispatches all buffered messages for each consumer at once. When the messages are buffered and dispatched later, the global ordering of messages is lost. For example in Figure 4-6, all the messages for the first consumer would be dispatched before dispatching any messages to second consumer. As a result, the messages M4 and M6 generated after M1 are dispatched to first consumer before the message M1 is dispatched to the other consumers. To avoid this lack of correct ordering, this design uses event sequence number to keep track of the order of arrival of messages at server. To ensure the messages are dispatched in the same order as they arrive at the server, before dispatching any messages the thread checks if all the messages with lower ESN are already dispatched.

Both the above-mentioned designs use separate log files for each consumer. When the main memory buffer (Consumer buffers/Object store) is full, consumer logs are updated such that, they reflect the information that there is an event in the log that has to be pulled to main memory when empty slots are available. A thread does this work of pulling from secondary storage to main memory. This thread adopts Round Robin method to ensure fairness among clients. To pull the messages that have to be delivered first, the thread has to pull the messages in order of their ESNs. This raises the following issues:

- Store and update the location (in which consumer log) of next ESN. This could complicate the process.
- According to the number of empty slots and message distribution, the process may have to do multiple file opens and closes to fill the object store. This would hinder the basic idea (concurrence) in introducing individual logs for clients.

However, the concurrency provided by multiple logs is lost. So to avoid the complicated process of keeping track of next ESN on secondary storage, we decided to introduce single log for all the events on the server. Hence, the current system uses a central object store wherein all the in-memory messages are stored and a single log file for messages that arrive onto the GED server for all the consumers.

4.4.4 Current Design

Figure 4-7 shows the refined design. There are two threads, Dispatch Thread and Pull Thread. One dispatches the messages in object store to the consumers of that message. The other pulls in the logged messages, if any, from the log into object store whenever there are empty slots available. To avail some concurrency between the threads that modify the object store, this design introduces two object stores (similar to the concept of double buffering), one stores the messages with odd ESNs and other stores the messages with even ESNs. Object store, as the name indicates is a place where all the incoming message objects are stored. It ensures the First In First Out (FIFO) order in message delivery. Hence, messages are dispatched in the order in which they arrive and thus maintaining the global ordering. As this is a shared data structure, it is protected with a lock.

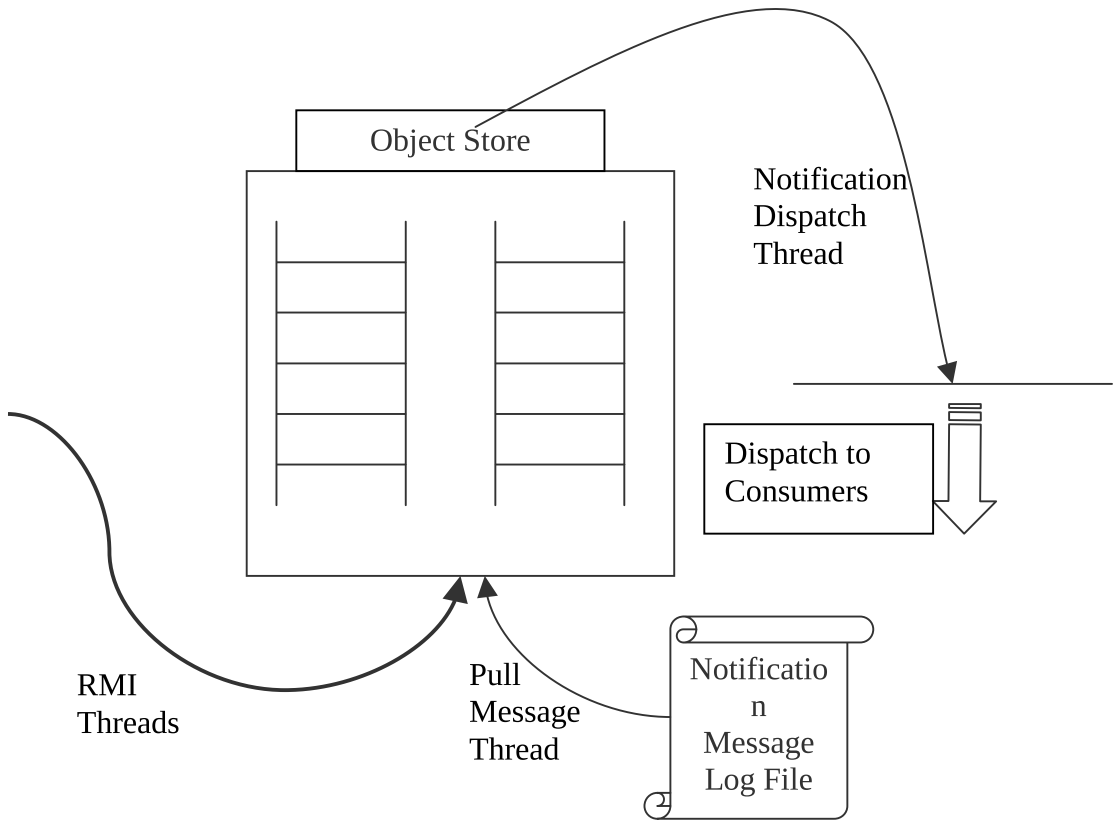


Figure 4-7 Refined Buffer Manager

4.5 Buffer Manipulations

When a new message arrives, if the GED is running in PERSIST mode, it is logged first and an attempt is made to add it to the main memory buffer i.e., object store. If it can be added, the buffESN is updated in the log file. If the main memory buffer is full, then the pendingEvts counter in the log file is incremented indicating that there is a logged event that needs to be pulled into main memory whenever there is an empty slot available for it. When a message is dispatched from the main memory buffer, it creates an empty slot. Any logged messages that could not be added to the main memory buffer on their arrival are then read from log file into main memory.

Clients send event notification messages to GED through RMI calls. The RMI threads resulting from these calls add the messages to object store. There are mainly two other threads, a dispatch thread and a pull thread. Dispatch thread is responsible for reading the messages from object store and dispatching them to the consumers and it takes care of any failures during this process. Pull thread is the one that pulls the logged messages, if any, into main memory whenever there are empty slots available in the main memory buffer i.e., object store.

The dispatch thread is notified when a message is added to the object store. Dispatch thread obtains the lock on the appropriate object store and extracts a message from the top of the queue. The consumer list for this message is read and the message is dispatched to all the consumers that are up, with an RMI call. The dESN counter in log is updated to represent the largest message ESN that is dispatched to its subscribers.

Whenever a message is dispatched by the dispatch thread, there is an empty slot created in the object store. This is notified to Pull Thread, which checks if there are any events pending in the log that can be brought into main memory. If there are any, according to the number of empty slots available, they are read from the log file and added to the appropriate object store based on their ESN. At the same time, the buffESN in log file is updated to represent the largest ESN present in the buffer.

4.6 Guaranteed Delivery of Events

Since the communication model is asynchronous, the GED wouldn't know whether a client is alive or crashed until it tries to communicate. Hence, while making an RMI call to the consumer, if GED receives an exception then it indicates a problem with the consumer. The problem might be that the consumer is slow to pick the messages or it has crashed. Java GED can support the slow consumers but cannot support the client crashes and recovery. GED has to make sure the client has crashed before declaring it crashed and at the same time it has to take care of the undelivered messages. To avoid loss of these undelivered messages GED introduces a buffer window such that even if the client fails to pick up a certain number of messages, they will not be lost. Once a client goes over the limit GED assumes that the client has crashed. Nevertheless, within the allowed window GED should accommodate the slow consumers.

If the number of message delivery failures to a client is within the buffer window size, the undelivered messages are stored. GED allocates a buffer of specified window size for each slow consumer. Whenever there is a new message for this consumer, GED appends it to the end of undelivered message list and tries to send them all in a single call. The size of the window or the number of messages the client can fail to pick is specified by the TIMEOUT variable in global configuration file.

Similarly, clients would not know if GED has crashed until they try to send a message and receive an exception. The messages that are to be passed between clients and server should not be affected due to the server failures.

Unaware of GED crash, client applications try to send event notification and event detection request messages to server. Messages sent to server when it was down will be lost. To avoid discrepancies in event detection due to loss of these messages, they should be logged on client side. GED should be able to receive undelivered messages from the clients when it recovers. If multiple events that have occurred over a period of time are received from each client are processed individually, the global ordering of the event occurrences will be lost. To maintain the global order of event occurrences, the event notification messages from different clients should be sorted according to the time stamp associated with them and processed in the same order, as they would be if sent when they are generated. During the recovery process GED queries each client for any messages it lost. It collects the logged messages, if any, from client applications. Assuming that all the client clocks are in sync, the event notification messages from all the clients are sorted based on the timestamps associated with the messages and processed as if they were received in that order. This ensures that they are processed in the same order they were generated, the only drawback being the delay in processing because of crash.

4.7 Extensions to Configuration File

A system requires certain information to initialize itself. For example, GED as a server requires information, such as the mode in which it should start, whether it should persist events and provide recovery etc. The users need to convey these constants and flags to the system to configure it to their requirements. Either these values can be passed as arguments to the program or a configuration file can be used to store this information. Providing this information as a configuration files has the advantage that the information is recorded in a file, which can be looked up or changed for each execution. Providing as arguments does not provide a record of usage and from the usage point of view is difficult to manage if the number of parameters were large. Hence, we use configuration files to allow the user to tailor the GED server and local applications. In addition, several different configuration files can be created and used for different applications in which we want different features from the same GED server. Configuration files in GED were introduced by [5]

The user, according to the requirements and available resources, can configure the system by setting the parameters in the file. The system reads the file during the process of initialization and configures itself accordingly. Global configuration file is used to configure the GED and application configuration file is used configure the client applications. The following sections discuss the extra parameters that have been added to the global and application configuration files. Figure 4-8 shows a sample global configuration file.

4.7.1 Extensions to Global Configuration File

In persist mode of operation of GED; the buffer manager manages the main memory buffers used to store the incoming messages and handles the buffer overflows. For this purpose, the user needs to convey the amount of main memory available to store the messages. Since the management is done at the message level, the user need to convey this in terms of the number of messages that can be stored in main memory. In a non-persistent mode of operation, buffer manager assumes the infinite availability that is limited by the amount of memory available to the Java Virtual Machine. User can supply this information using the BUFF_MAX property in the configuration file.

Persistence of events and recovery capability adds an overhead to the system, which in turn affects the system performance. Hence, the user must be given option to turn this capability off if he/she does not want to provide recoverable capability to their system. Configuration file provides this with PERSIST property.

Log file is nothing but a sequential stream of message objects. To provide the capability to navigate through and provide non-sequential access of message objects stored in log file, index table is added to the notification message log file. As discussed earlier this puts a limit on the log file size. Number of messages that can be stored in a log file is the number of index records that can be held in the index table. This is conveyed to system through the LOG_SIZE property of the configuration file.

Along with notification message log file, there are several other log files, which store the system state from time to time. All these log files are stored in a Log directory. LOG_DIR property gives the path to this directory.

GED server can be started afresh or it can resume from the previous state (either after a crash or a proper shutdown) if appropriate log files exist. The mode in which the GED is started is conveyed through the MODE property of configuration file.

TIMEOUT property indicates the number of message dispatch failures that can be tolerated before declaring that a consumer has crashed or there is a network failure. This will be discussed in detail in section 4.6.

```
BEGIN
PERSIST TRUE
LOG_DIR .\..\Log\
BUFF_MAX 100
LOG_SIZE 10000
GED_NAME ged1
MODE INIT
TIMEOUT 5
MAPPING consumer_newdelhi consumer_seoul
MAPPING producer_newdelhi producer_seoul
MAPPING conprod_newdelhi conprod_seoul
MAPPING airTrackApp_myhome airTrackApp_seoul
MAPPING shipTrackApp_myhome shipTrackApp_seoul
MAPPING control_myhome control_seoul
END
```

Figure 4-8 Global Configuration File

4.7.2 Extensions to Application Configuration File

In accordance with the recovery capability added to the GED server, there are certain things that are added at the local application level.

The messages produced by the producers when the server is down, cannot be sent to the server. The client must store them so that they can be sent when the server comes up and asks for them. As with any other logging process this involves some overhead and the user is given an option to turn this capability off with LOCAL_LOGGING property in the application configuration file.

LOG_DIR property indicates the place where the above-mentioned log file should be stored. Similar to the server, the clients can also be started in initialize or resume mode. This facility is provided by the MODE property in the configuration file.

4.8 LOCKS

A situation where several threads access and manipulate the same data concurrently, and the outcome of the execution depends on the order in which, the access takes place, is called race condition. To avoid race conditions, only one thread at a time should be able to manipulate the shared data. To make such guarantee, some form of synchronization mechanism is needed. Locks are used to synchronize the access of multiple threads to shared data structures.

The util.concurrent package provided by [19] is used as the base model. [19] provides a high-level design principles and strategies, technical details surrounding constructs, utilities that encapsulate common usages, and associated design patterns that address particular concurrency problems. util.concurrent package provided with [19] comes with lot of functionality. It provides a variety of locks used for synchronization. This thesis has extracted only the necessary classes and modified them according to our requirements. This implementation adopts the acquire/release protocol provided by the "sync" interface. Three types of locks are introduced here, Mutex, ReadWrite and Semaphore.

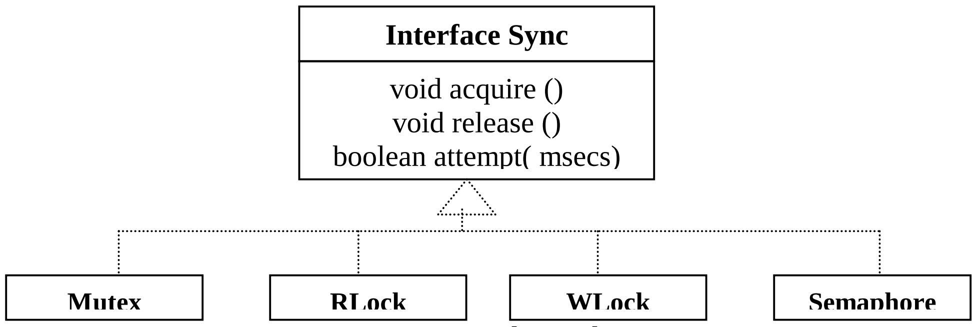


Figure 4-9 Locks package

4.8.1 Mutex

Mutex is a short form of Mutually Exclusive Object. Mutex is a synchronization variable that has only two states, locked and unlocked. The first thread that locks the mutex gets ownership to data and any subsequent attempts to lock the data will cause that thread to go to sleep. When the owner unlocks it, one of the sleepers will be awakened and given a chance to obtain ownership. When a program is started, it creates a mutex for a given resource at the beginning. After that, any thread needing the resource must use the mutex to lock the resource from other threads while it is using the resource. Mutex is suitable for the situations that need most restrictive access to data because it allows only one thread to access the data.

4.8.2 ReadWrite

In situations where data is read more frequently than it is written, ReadWrite lock is used. This lock is composed of two locks, read lock and write lock. ReadWrite lock allows multiple threads to read lock the data concurrently but only one writer can acquire the write lock at a given time. Any number of read locks can be issued, so long as there are no writers. Write lock is mutually exclusive.

4.8.3 Semaphore

Semaphore is a synchronization variable that has a value. It can be incremented to an arbitrarily high value but can be decremented only to zero. The value of a semaphore is the number of units of the resource that are free. If there is only one resource a binary semaphore, with values zero or one is used. Semaphore operations are known as P and V operations. The P operation attempts to decrement the variable and thus claim the resource. V is the inverse, it increments the semaphore. It simply makes a resource available again after the process has finished using it. If semaphore is greater than zero, P operation succeeds; if not, the calling thread must go to sleep until different thread increments it. Semaphore is initialized before any requests are made. This package uses acquire/release protocol. General usage of locks looks as shown in Figure 4-10.

```
try{
        lock.acquire();
        try {
                action( );
        }
        finally {
                lock.realse( );
        }
}
catch( InterruptedException ie ) {
        /* response to thread cancellation during acquire */
}
```

Figure 4-10 Lock usage

Mutex locks are used to synchronize access to shared data structure on GED server. When the GED server recovers from a crash, it issues a lock called *recoverLock* that locks the entire recovery process, and releases the lock only when the recovery is over. *RecoverLock* is a mutex lock. GED recovery process is treated as atomic and access to all the data structures is restricted to the main thread. This ensures that, threads resulting from client RMI calls and other threads in GED cannot access the shared data during the GED server recovery. Similarly, the access to object stores in buffer manager module is synchronized using mutex locks. Any access to these object stores results in an addition of new message to it or a deletion of message from it. Hence, mutex locks are used for this purpose. Notification message log file is also locked using a mutex lock. Every access to this log file, either a read or write of a notification message results in update of the header information (buffESN, dESN, pendingEvts, index table) associated with the log file.

4.9 Summary

This chapter explains the design of buffer manager, persistence and recovery of GED. It first discusses the persistence of the event information and state of the GED server. It explains the design of GED recovery. It then goes into the design of the buffer manager and buffer manipulations. It discusses the extensions that have been made to the global and application configuration files. How GED guarantees the delivery of events is explained and finally it discusses the locks used to synchronize access to shared data structures.

CHAPTER 5

Implementation of Persistence

This chapter discusses the implementation details of persistence. The structure of this chapter is as follows: It first goes into the implementation details of the data persistence using Java Object Serialization and implementation of logs. It then explains the buffer manager module and the threads involved in it. It explains the locks used to synchronize different data structures in the implementation and finally it explains the configuration of the client applications and GED server.

5.1 <u>Implementation of Log Files</u>

Writing and reading of files involve considerable overhead in terms of opening the file and closing the file. In addition, sequential access (typically used) is not appropriate as we are using the log files to bring events into the buffer. This is different from how log files are typically used by a DBMS for recovery. During recovery, since all the log records need to be read, sequential access is fine. In our case, we need to read only a few selected records both for buffer management and for recovery. Hence, the main emphasis here is to minimize the file IO involved in writing to and reading from logs. For the log files from which we need to do a non-sequential read an indexing mechanism is needed. Java Serialization is used to persist the data values. The log files are written in append mode. Instead of persisting a data structure, as it is whenever it changes, GED persists only the information that reflects the change made to its state. This information is appended at the end of corresponding log file.

5.1.1 <u>Basics of Java Object Serialization</u>

Java object serialization [17] provides the ability to write or read java objects to and from a byte stream. It allows Java objects and primitives to be encoded into a byte stream suitable for streaming to a network or to a file-system. The Java Serialization API provides a standard mechanism for developers to handle object serialization. The API is small and easy to use.

To persist an object in Java, the object must be serializable. Object is marked serializable by implementing the java.io.Serializable interface or by inheriting that implementation from its object hierarchy. Serializable interface has no methods, so the object itself need not implement any methods. This interface signals the Java virtual machine to use default serialization mechanism to serialize this object. Serialization and de-serialization is done through the ObjectInputStream and ObjectOutputStream respectively. ObjectOutputStream class provides writeObject method. This method is responsible for saving the state of the object. The readObject method provided by the ObjectInputStream class is responsible for restoring the object from the serialized byte stream.

The following example shows the serialization of the Person object to a file and de-serialization of it from the file. WritePerson class serializes the Person object to a file. This class creates an ObjectOutputStream for the Person object and writes it to a FileOutputStream named Person.ser. It formats the object as a stream of bytes and saves it in the Person.ser file. ReadPerson class de-serializes the serialized Person object. It creates an ObjectInputStream from the FileInputStream, Person.ser. It reads the byte stream from Person.ser and reconstitutes the Person object from it. The code sample is shown in Figure 5-1

Figure 5-1 shows a very concise and easy way to implement serialization. The same code can be used to persist a complex object as the serialization mechanism works by transitive reachability. Reachability means that all the objects reachable from this object will also be serialized. If the object passed to the writeObject method contains references to other objects, the passed object and the other objects reachable from it will also be serialized. Java Object Serialization handles cyclic graphs. Each object visited is marked. If a cycle exists and an object is visited again, the mechanism knows that this object has already been serialized. Therefore, it only puts enough information into the serialized form so that the cycle can be rebuilt when the data is de-serialized.

```
import java.io.*;
public class Person implements Serializable {
    public String name;
    private String password;

    public Person(String name, String password) {
        this.name = name;
        this.password = password;
    }
}
```

```
class WritePerson {
    public static void main(String [] args) {
        Person p = new Person("Fred",
"cantguessthis");
        ObjectOutputStream oos = null;
        try {
            oos = new ObjectOutputStream(
            new FileOutputStream("Person.ser"));
            oos.writeObject(p);
        }
        catch (Exception e) {
            e.printStackTrace();
        }
        finally {
            if (oos != null) {
                try {oos.flush();}
                  catch (IOException ioe) {}
                try {oos.close();}
                  catch (IOException ioe) {}
            }
        }
    }
}
```

```
class ReadPerson {
    public static void main(String [] args) {
        ObjectInputStream ois = null;
        try {
            ois = new ObjectInputStream(
            new FileInputStream("Person.ser"));

            Object o = ois.readObject();
        }
        catch (Exception e) {
            e.printStackTrace();
        }
        finally {
            if (ois != null) {
                try {ois.close();}
                 catch (IOException ioe) {}
            }
        }
    }
}
```

Figure 5-1 Code showing serialization process

5.1.2 Tailoring Serialization

In Java, object serialization is done through *writeObject ()* method provided by the ObjectOutputStream class. As explained in previous section, to serialize objects into a file, a FileOutputStream would be used. To serialize objects into a file in append, the application should be able to skip through the file to its end and then add the newly serialized object byte stream by using the *writeObject ()* method. Nevertheless, neither ObjectOutputStream nor FileOutputStream classes allow the programmer to browse/skip through the serialized byte stream in file.

In Java, object de-serialization is done through *readObject ()* method of ObjectInputStream class. To restore data from a log, the recovery process should be able to read the objects and use them in restoring the data values. In some situations different types of objects are written into the same file in append mode. Trying to read such objects using the *readObject ()* method will result in StreamCorruptedException because, ObjectInputStream cannot demarcate between the byte streams of objects of different types. It cannot demarcate end of one object and start of another object.

In case of notification message log file, explained in chapter 4, the objects at a specific position should be restored by de-serializing the serialized byte stream, corresponding to that object, in the file. This needs the implementation to browse/skip through the serialized byte stream in file, extract the serialized bytes corresponding to this object from the file and de-serialize them to restore the object. Even if the application has to read the objects consecutively serialized into the file, it has to browse through the file and de-serialize each object one by one. ObjectInputStream or the FileInputStream classes does not allow browsing /skipping through the serialized byte stream. Simply trying to skip through the file to a specified byte position and using the *readObject ()* method would result in a StreamCorruptedException.

To overcome these problems a mechanism should be devised to handle the serialized byte stream according to our requirements rather than depending on the ObjectOutputStream and ObjectInputStream classes. Serialization and de-serialization mechanism should be dealt at byte level rather than at object level. The application should be able to write the serialized objects at the end of the file. If the size of the byte stream of a serialized object can be calculated, application can demarcate between the objects. During the serialization of an object, offset of the serialized bytes in the file and the number of bytes in serialized stream should be marked. To restore an object from file, file pointer should be skipped to the starting position of its serialized bytes and the corresponding number of bytes should be de-serialized.

The solution to above problems is achieved by using ByteArrayInputStream and ByteArrayOutputStream classes in conjunction with RandomAccessFile class. According to the Java documentation [20], "*A random access file behaves like a large array of bytes stored in the file system. There is a kind of cursor, or index into the implied array, called the* file pointer*; input operations read bytes starting at the file pointer and advance the file pointer past the bytes read. If the random access file is created in read/write mode, then output operations are also available; output operations write bytes starting at the file pointer and advance the file pointer past the bytes written. Output operations that write past the current end of the implied array cause the array to be extended. The file pointer can be read by the* `getFilePointer` *method and set by the* `seek` *method.*"

RandomAccessFile class provides enough functionality to read, write and browse through a file. Using methods provided by this class, the serialized bytes can be written at the end of the file, thus achieving our requirement to write log files in append mode. It provides functionality to seek to a particular position in the file and read specified number of bytes from the file, thus solving our problem of extracting the serialized bytes of a particular object.

To find the size of the serialized byte stream of an object ByteArrayOutputStream class is used. According to Java documentation [20], *"This class implements an output stream in which the data is written into a byte array. The buffer (byte []) automatically grows as data is written to it. The data can be retrieved using* `toByteArray()` *and* `toString()`*."* The size of the byte array resulting due to serialization of an object is not known before hand. But, after serializing it into a ByteArrayOutputStream, the `size()` method provided by this class can be used to find out the size of the buffer (byte[]) and thus the size of the serialized byte stream.

Similarly, to de-serialize the byte array read from the file (using RandomAccessFile), ByteArrayInputStream is used. Through this class, the application supplies the byte array to ObjectInputStream and invokes readObject( ) method to restore object.

```
byte[] barr = null;
int objsize;
try {
   bos = new ByteArrayOutputStream();
   so = new ObjectOutputStream(bos);
   so.writeObject(obj);
   objsize = bos.size();
   barr = new byte[objsize];
   barr = bos.toByteArray();

   rf.seek(offset);
   rf.write(barr);
   bos.close();
} catch (Exception e) {
   System.out.println("Error in Logwrite");
   System.out.println(e);
  }
  finally {

   if (so != null) {
     try {
        so.flush();
        so.close();
     } catch (IOException ioe)
     {ioe.printStackTrace();}
```

```
byte[] barr = new byte[(int)objsize];
try {
   rf.seek(offset);
   int r = rf.read(barr);
   bis = new ByteArrayInputStream(barr);
   si = new ObjectInputStream(bis);
   obj  = si.readObject();
} catch (Exception e) {
        System.out.println(e);
   }
  finally {
   if (si != null) {

     try {
        si.close();
     } catch (IOException ioe)
     {ioe.printStackTrace();}
    }
  }
```

Figure 5-2 Code showing Serialization into a ByteArrayStream

For cases where the objects to be persisted can be directly written to and read from files, they are serialized or de-serialized directly using the File Input/Output Stream classes as shown in Figure 5-1. However, for situations explained above serialization and de-serialization processes are done using ByteArray Input/Output Stream and RandomAccessFile classes. A small code snippet of the process is shown in Figure 5-2

5.1.3 Notification Message Log File

Notification message log file stores the notification messages along with the information required for the buffer management and recovery of the main memory buffer i.e., Object Store. First twenty bytes of the file are allocated to store three integers and a long value. First four bytes represent buffESN, the largest ESN in the buffer. Next four bytes store dESN, the largest ESN that has been dispatched to and received by the consumer. Bytes from four to eight store pendingEvts, the counter that reflects number of events that are waiting in the log file to be pulled into main memory buffer. Following eight bytes store the index table size represented as a long value. The size information of index table, itSize, is stored to make the reads and writes of index table data structure easy. The itSize indicates the size of serialized byte array of index table. The rest of the file contains the serialized message objects.

Index table as shown in Figure 5-3, is nothing but a collection of table elements. Each element has three fields: ESN, ObjectSize and Offset. ESN stores the unique sequence number assigned to the message when it arrives on the server. ObjectSize represents size of the serialized byte stream of notification message object corresponding to id in first field. Offset represents byte position in the file at which the byte stream of this message object starts. Number of elements in index table is specified by the ITSIZE property of configuration file.

Index Table		
ESN	Object Size	Offset
⋮		

Figure 5-3 Index Table

If the user opts for PERSIST mode of operation, GED creates the log file. It instantiates the index table of size given by ITSIZE and inserts it into the log file. All the notification messages coming to the GED are written to log file before being added to the object store. The message objects are serialized into a byte array. During the process, byte size of the serialized object is inferred from the byte array size. This byte array is written at the end of the log file, the position of which is calculated from the length of the file. After writing the object byte array into the file, GED reads the index table into main memory, updates it with the information of the message object size and its offset in the file and writes it back.

To extract a message object with a specific ESN from the log, index table is read in to main memory. It is queried with the ESN to find the position and size of the object. GED then seeks to that point in the file, reads the required number of bytes into an array and de-serializes them to obtain the requested message object.

5.1.4 Other Log Files

To restore the GED from a crash, all the information needed for recovery must be in stable storage at the time of recovery. GED needs to log all the required data structures. All the data structures that are logged for this purpose are explained in detail in chapter 6. Each of these data structures is stored in separate logs. The name of the log file is used to correlate it to the corresponding data structure. All the log files are stored in a directory specified by LOG_DIR property in configuration file. As mentioned earlier, the importance is given to reduce the amount of logging that is done to restore the data structures. The data that is persisted for each data structure and the process of restoring them from log files is discussed in detail in chapter 6.

5.1.5 Log Compression

Notification message log file provides the capability to index into the byte stream and extract a specific notification message. As explained earlier, this capability comes at the cost of limiting the file size. Log file size is limited by the number of elements of the index table used in the file. This attribute, ITSIZE, is read from the global configuration file at the time of GED initialization. If the log reaches this limit, further logging would result in a failure. To overcome this, GED provides a functionality to compress the notification message log. At any point of time, the dESN indicates the maximum ESN that was successfully dispatched to its consumers. As all the messages with ESN less than dESN are already dispatched, they don’t need to be kept in log any more, so they become the potential candidates for purging.

As this file is a stream of bytes corresponding to the serialized message objects, if an application wants to delete a message object it has to first delete the bytes corresponding to this object and then move the remaining bytes forward to reclaim the empty space in the file. Thus deleting these serialized message objects would change the starting positions of all other messages in the log file. The Index table should be updated accordingly to reflect the latest start positions.

Messages with ESN less than dESN can be deleted from the log. For each message that can be deleted, size of its serialized byte stream is fetched from index table and the sum of these byte stream sizes would give us the number of bytes that can be removed from the file. The new start positions of each of the messages that will be left in the file after compression is calculated. The information about the new start positions along with the ESN and object size information in index table is recorded into a new index table. This index table, along with the buffESN, dESN and pendingEvts counter information is written to a temporary file. The bytes that will be left after removing bytes corresponding to the deleted objects into can now be copied into this temporary file. Now the temporary file reflects the state of log file after deleting the unwanted messages. By deleting the current notification message log file and renaming the temporary file to the notification log file name, the original log file is replaced with the compressed log file. Thus, the compression is achieved.

As this process is updating the contents of the log file, during the entire process the log file is locked. Any new messages coming during this time will just wait on the thread to complete the compression process. When 80% of the log is filled, GED initiates the log compression. The compression can also be initiated as a separate client process.

5.2 Implementation of Buffer Management

In the existing implementation discussed in chapter 3, all the event information is kept in main memory and sent to clients. It assumes unlimited memory availability. All the incoming messages are stored in a Vector. If memory is not sufficient in GED, events are lost. This memory overflow may even result in a system crash. As the communication is asynchronous, when a consumer is not responding/slow, the producer will still send events; these events are lost due to the lack of main memory buffers.

To overcome these limitations, a buffer manager has been added to the main memory implementation. This module should manage the main memory used to store event notification messages. It should be able to handle the buffer overflows and the required read/write access to secondary storage.

Buffer manager manages the notification message buffer called object stores. It provides a mechanism to add messages to these object stores, retrieve messages from the stores, and dispatch them to their consumers. As this module is responsible for dispatching the messages to the consumers, it also accommodates the slow consumers. If GED is running in persist mode it handles the required read/write access to secondary storage.

The threads involved in buffer management are RMI threads that bring in messages from clients, dispatch thread that handles the message dispatch from the buffer to its consumers and pull thread that pulls messages from log file into main memory buffer. Figure 5-4 shows the object store and the threads that access it.

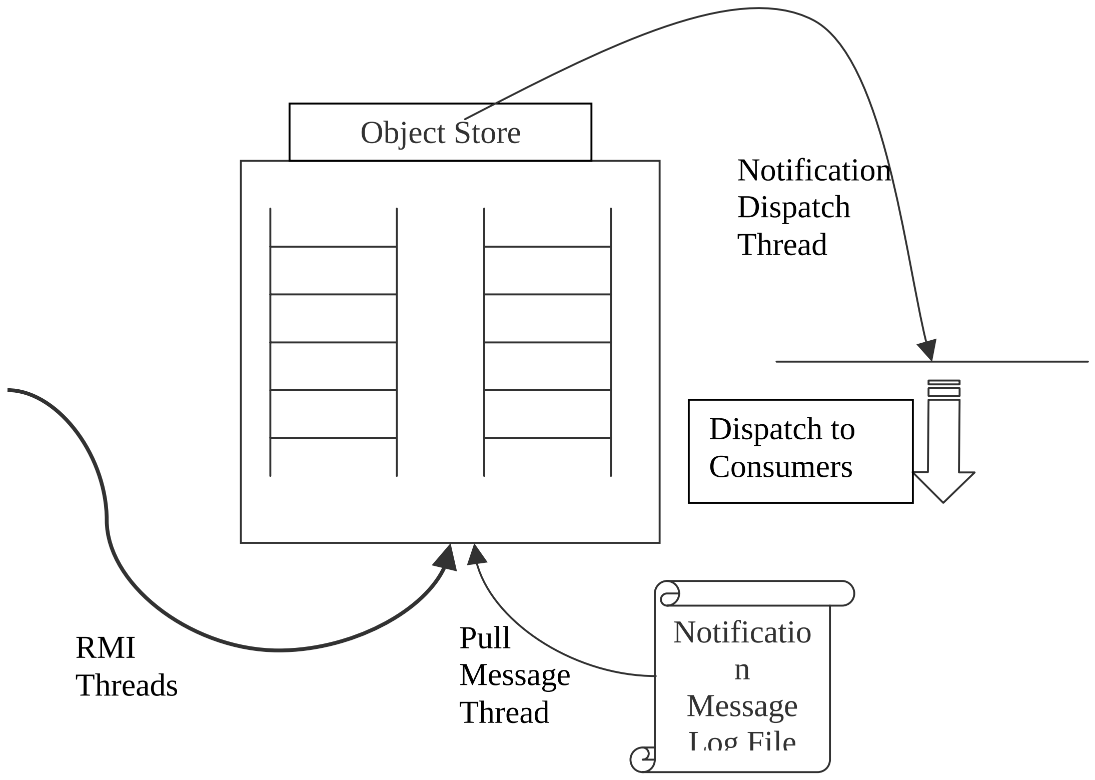


Figure 5-4 Buffer manager

5.2.1 Object Store

Object store is the place where all the incoming event notification messages are stored. As mentioned earlier in chapter 4 global ordering of the message dispatch is important to ensure the correct event detection semantics. The main requirement of this data structure is that it has to ensure that the messages that arrive first have to be dispatched first. Hence, it has to be a queue data structure with First In First Out (FIFO) policy. Any new message is added at the end of the queue and the dispatches are done from the top of the queue.

Object store is accessed by the threads that try to add and remove messages from it. The access should be synchronized and a mutex lock is used for the purpose. Any thread trying access the object store should wait to acquire this lock. This deprives of any concurrence between the different threads. Some amount of concurrency can be achieved by using two object stores, a concept similar to double buffering. The object store is divided into two parts, one that stores messages with odd ESNs and the other that stores messages with even ESNs. This approach provides concurrency between the threads that access the store.

In non-persistent mode of operation, the object store size is limited by the amount of the memory the JVM can allocate to this data structure. In a persistent mode of operation as the buffer manager is responsible to manage the object store, it needs to know the maximum number of messages that can be stored here. This is indicated by the BUFF_MAX property in global configuration file.

5.2.2 Notification Dispatch Thread

This thread extracts the notification messages from the object store and dispatches it to its consumers. Initially, it waits for notification from buffer manager. It is notified when there is a new message in the object store. GED stores a mapping between the global event name and all the consumers for that event. For each event notification message it extracts from the object store, it gets the consumer list from the mapping. Before sending the message to consumer, this thread checks if this is a slow or crashed consumer. A slow or crashed consumer would have not picked up earlier messages. As discussed earlier in section 4.6, buffer manager allocates special buffers for slow consumers and stores the undelivered messages in these buffers. Figure 5-5 shows the slow consumer buffers data structure. A consumer is declared crashed if the number of message dispatch failures to it goes over the limit specified by the TIMEOUT variable in configuration file. To avoid unnecessary communication cost, no messages are sent to the crashed clients. For example, in the Figure 5-5 Cons_Bangkok is a slow consumer and Cons_Paris is a crashed consumer (Assuming TIMEOUT = 5). Once the consumer crash is detected, its slow consumer buffer is cleared and just the first element indicating that this consumer has gone over the limit is stored. If the consumer is a slow consumer, all the undelivered messages are extracted and the current message is added at the end of the list. Notification dispatch thread then tries to send this list to the consumer. If there are no pending messages for this consumer, notification dispatch thread tries to send only the current message to the consumer. If the communication intended for a slow consumer is successful, its consumer buffer is cleaned and its entry is deleted from the data structure. If this communication fails, the message is added to the consumer buffer and the counter information is updated accordingly.

Slow consumer buffers are implemented using Java hashtable data structure. The key is consumer id and the value corresponding to this is a Vector storing the messages that were not received by the consumer. The first element of this Vector stores the count of such messages. Number of messages that can be stored in each consumer buffer is limited to specific amount mentioned by the TIME_OUT variable in configuration file.

At the end, this thread updates the dESN field in the log file with the latest ESN dispatched.

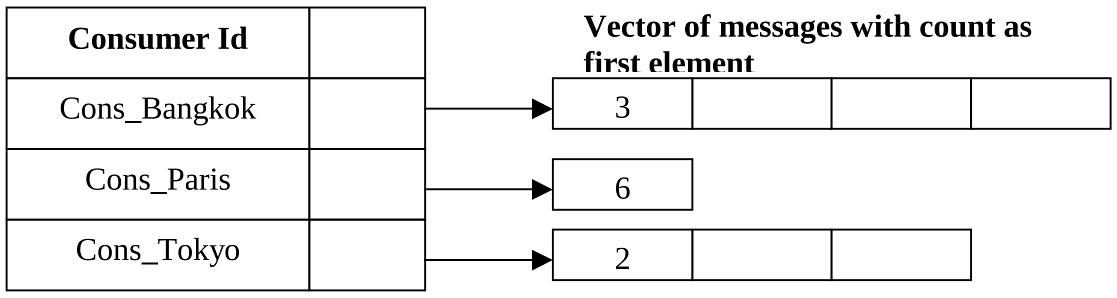


Figure 5-5 Slow Consumer Buffers

5.2.3 Pull Message Thread

This thread exists only when the GED is running in persistent mode. It handles the reads from log file. This thread reads the serialized messages from the log and puts them in the appropriate object store according to its ESN. Initially, it waits for the notification from the buffer manager. When the notification thread dispatches a message, it creates an empty slot in the object store. As and when an empty slot appears in one of the object stores, the buffer manager sends a notification to pull message thread.

Number of messages that could not be added to object store and are to be pulled into main memory is indicated by the pending events counter (pendingEvts) in log file. Once the pull message thread is notified from buffer manager, it checks the pending events counter. If there are any, it reads one message at a time and puts it in appropriate object store. It does so until the object store is full or there are no more pending events. It updates the buffESN field and pendingEvts counter in log file to appropriate values after each message read it does.

5.3 Implementation of Recovery and Other Locks

As mentioned earlier util.concurrent package [19] is used to implement locks. This provides three different types of synchronization protocols. They are:

Sync: acquire/release protocols

Channel: put/take protocols

Executor: executing Runnable tasks

The Sync interface is used to implement Mutex, ReadWrite and Semaphore locks. Interface Sync provides three methods. First, *acquire ()* is the operation performed to enter into the synchronized block. Second, *release ()* is the operation performed to exit the synchronized block. Third, *attempt ()* returns true only if the lock is acquired within the specified time. Mutex, ReadWrite and Semaphore classes implement this interface. These method implementations are taken from the util.concurrent package.

Two additional functionalities are added to Mutex. These are *isAvailable ()* and *waitUntilAvailable ()*. *isAvailable ()* method returns true if lock is available at the given time or false if someone is holding the lock. "*waitUntilAvailable ()*" method does not acquire the lock but it makes the thread that invoked it to wait until the lock is available. The use of these methods is explained in the next section.

ReadWrite lock comes with the facility to control the number of readers and writers. It also provides mechanism to assign priorities to readers and writer. Our implementation is stripped off all this functionality. The classes corresponding to this lock are modified to provide the bare minimum functionality of issuing read locks and write locks because our requirements do not need this additional functionality.

Semaphore lock implementation is retained as provided by the package.

5.3.1 Implementation of Recovery Lock

During the process of recovery, GED registers its remote communication interfaces with the RMI registry before the recovery is complete. This opens the GED for incoming messages from client applications, even though the GED has not completely recovered. Processing of these incoming messages would involve modifying the incompletely recovered GED state. Hence, the processing of such messages should be deferred until the recovery is complete. This is achieved by using the *RecoveryLock*. RecoveryLock is to lock the entire recovery process and release the lock only when recovery is over. In the normal operation, the threads would check for the availability of this lock but not acquire it. Holding this lock ensures that others cannot access the shared data during GED server recovery. Figure 5-6 shows the pseudo code for recovery lock algorithm.

When GED Recovers:	When server accesses to the data structure:
Obtain Recovery Lock Communication layer recovery Global event graph recovery Buffer Recovery Release Recovery Lock	If (recovery lock is available) Do not obtain the recovery lock, but obtain the individual lock on the data structure

Figure 5-6 Recovery Lock

5.3.2 Other Locks

Object stores in buffer manager module are exposed to multiple threads. They will be accessed by the RMI threads emerging from client calls to server, notification dispatch thread and pull message thread. RMI threads and pull message thread add new messages to the object stores, while the notification dispatch thread removes messages from stores. All the threads access result in structural modifications of these data structures. Hence, mutex locks are used to synchronize access to object stores. Each store access goes through acquire and release of the mutex lock associated with it.

Log file is also exposed to the same threads as the object stores. RMI threads adds new messages to the log file and pull message thread reads the serialized messages from log. Each of these access results in modification of header information (buffESN, dESN, pendingEvts) associated with the log file. Notification thread updates the dESN field after dispatching a message to its consumers. Log file is prone to race conditions. Hence, mutex lock is used to synchronize access to log file.

5.4 Configuration File

Configuration files are used to convey the setup information to the system and customize it according to user requirements. These constants are read into static variables at the time of system initialization. The configuration file location and name either can be passed to the application as a command line argument or could be a standard name in a standard location so that the application automatically locates it. As each execution of the application uses only one configuration file, it is associated with a standard name. The application configuration file is named as *App.config* and is located in the directory from which the application is run. Similarly, the global configuration file is named as G*lobal.config* and is located in the source file directory.

Table 5.1 shows the information specified in global configuration file (Global.config) and their default values. This file is read using the Property class in Java. Table 5.2 shows the information specified in application configuration file (App.config) and their default values.

Flag	**Description**	**Values**	**Default**	**Remarks**
GED_NAME	Name of the GED	String	GED1	
MODE	Start mode of the application	INIT RESUME	INIT	INIT – initialize, starts the application a fresh RESUME – resume, starts the application in resume mode. Uses the log files (if available) to recover the previous state.
PERSIST	Indicates whether to do persist the event information or not	TRUE FALSE	TRUE	ON – logs the event information. This includes notification messages, detection request messages and data structures needed to restore the state when started in RESUME mode. OFF – no logging.
LOG_DIR	Indicates the directory path for log files	Path name	Log dir in the distribution folder	Can be absolute or relative.
BUFF_MAX	Maximum number of messages that can be held in memory	Integer	100	Set according to the user requirements and available resources
LOG_SIZE	Maximum number of messages that can be stored in notification message log file	Integer	10000	Set according to the user requirements and available resources
TIMEOUT	Dispatch failure window size for consumers	Integer	5	User can specify the number of message dispatch failures that can be tolerated before declaring a consumer to be crashed.
MAPPING	Mapping from an old application ID to a new application ID.	String Values		Mapping allows the applications to be executed on different machines without changing and re-compiling the application.

Table 5.1 Global Configuration File

Flag	**Description**	**Values**	**Default**	**Remarks**
SCOPE	Scope of the application	LOCAL GLOBAL	LOCAL	For the stand-alone application To communicate with GED and participate in global event detection
RULE_SCHEDULER	Indicates whether to use rule scheduler or not	OFF ON	OFF	LED can trigger rules according to its priority and coupling mode. Rules are executed in the order in which they were defined for each event.
LOCAL_LOGGING	Indicates whether to do local logging or not	ON OFF	OFF	ON – logs information locally. This includes notification messages that could not be sent to server and data structures needed to restore the state when started in RESUME mode. OFF – no logging.
LOG_DIR	Indicates the directory path for log files	Path name	Log dir in the distribution	Can be absolute or relative.

Table 5.2 Application Configuration File

5.5 Summary

This chapter explains the implementation details of the data persistence and buffer management in GED. It first discusses the implementation of persistence using Java object serialization, problems with the basic approach and the solutions adopted to overcome those problems. It explains the implementation of buffer manager module, which involves the implementation details of main memory buffers and the threads that are involved in the buffer manipulations. The implementation locks used by GED are discussed. Finally, it summarizes parameters and their default values in the global and application configuration files.

CHAPTER 6

Implementation of Recovery

The effects of all updates must be durable: persistent despite system failures. A system failure results in the loss of the contents of volatile storage. After a recovery from system failure, all data values must reflect the stable state before the failure. Furthermore, all the information needed for recovery must be in stable storage at the time of recovery.

GED server can be mainly divided into three parts: The communication layer between the clients and the server, construction of event graph and detection of global events, and managing the buffer space available to store the notification messages. Thus, recovery of GED mainly involves recovery of these three parts.

1. GEDInterface
2. GlobalNodeManager
3. BufferManager

Even when the GED is down, the client applications will still be up and generating new event notification messages. Locally logging the messages and sending them to GED when it recovers can avoid the loss of these messages. The option to log these messages locally is given to user as logging involves some overhead associated with file IO. After restoring the state prior to crash, GED has to get these messages, if any, from clients and process them before continuing to normal operation.

The sections of this chapter explain in detail the data that is logged and recovery process of data structures in each of the three objects mentioned above. This chapter will also cover the processing of lost messages by GED because of the crash.

6.1 GEDInterface

GEDInterface provides a thin layer of communication between the clients and the GED server. Its attribute, ServerConnecterImp, enables the GED server to provide the register and un-register functionalities. It also implements SentinelComm interface, which enables it to forward the event detection requests and event notification messages to its clients. GEDMesgRecvImp provides the functionality required to receive the event detection requests and event notification messages from the clients.

Attributes:

- ServerConnectorImp
- GEDMesgRecvImp
- GlobalEventFactoryImp
- Hashtable prod_DectectnReqstHt
- Hashtable glbEvntName_consumerList
- Hashtable clntAddrsHt

ServerConnectorImp, GEDMesgRecvImp, GlobalEventFactoryImp are the remote interfaces used for the communication between the GED server and clients. These interfaces are registered in the RMI registry with specific names known to the clients. They are re-instantiated at the time of recovery and bound to RMI registry with the same names. This ensures that the clients can look up these latest communication interface objects at any time. Rest of the three Hash tables are logged and recovered from log. The following sections explain this in detail.

6.1.1 Client Address List (Hashtable clntAddrsHt)

This data structure maps client ID to its IP address. At the time of client registration, GED records application ID and its address into this address book (*clientAddrsHt*). This facilitates in looking up the client remote object in the RMI registry in order to make a remote invocation to a particular client later.

Key: String ConsId

Value: String ConsIP

Logging: filename: clntAddrsHt.log

Keep appending the String (“a / d” + “,” + appID+","+clientHost) object to the end of the file for each client that registers with GED server. The first character would indicate whether it’s a client registration or un-registration. An “a ” means the client has registered and a “d” means the client has unregistered.

Reason: Instead of serializing the entire hashtable, serialize only the necessary information to reconstruct this data structure (hashtable). To restore the state of the hashtable key-value pairs are needed. Hence, just serialize necessary key and value objects into a file in append mode. It is always good to minimize the number of objects to be serialized and de-serialized because it reduces the IO. As the key and value in this hashtable are strings, they can be appended to form a single String object and serialize it. The CPU time involved in appending the strings before serialization and parsing the string after de-serialization is lot less than the IO time involved in reading an extra object from the byte stream in log file and de-serializing it.

Recovery: Recovery process needs to rebuild the address book of clientIDs with their IP addresses. The log file stores the necessary information in custom serialized string objects in the clntAddrsHt.log file. The recovery process is as follows:

1. Instantiate clntAddrsHt hashtable.
2. De-serialize the String (“a / d” + “,” + appID+ “,”+clientHost) objects in sequence.
3. Parse it and get client ID, client IP and whether it’s a registration or un-registration.
4. Invoke put or remove (clientId, clientIp) method on clntAddrsHt.

6.1.2 Producer Event List (Hashtable prod_DectectnReqstHt)

This data structure associates producers with the list of detection request messages for them from the consumers. It contains producer ID and a vector of detection request messages. If it has an event detection request for the producer, the GED server will forward the message to producer.

Key: producerId (String)

Value: DetectionRequestList – vector of detectionRequestMessages

Logging: filename: prod_DectectnReqstHt.log

Keep appending the detectionRequestMessage objects to the end of the file.

Reason: The objective is to reduce the number of objects that should be serialized and de-serialized. Here the “key” object is producer Id that is a part of the “value” object. Instead of serializing both key and value pair, only the value object is serialized, thus reducing the amount of serialization and de-serialization done. The recovery process can get the key from the value object that is read (de-serializing the byte stream) from the file and insert the key - value pair into the table.

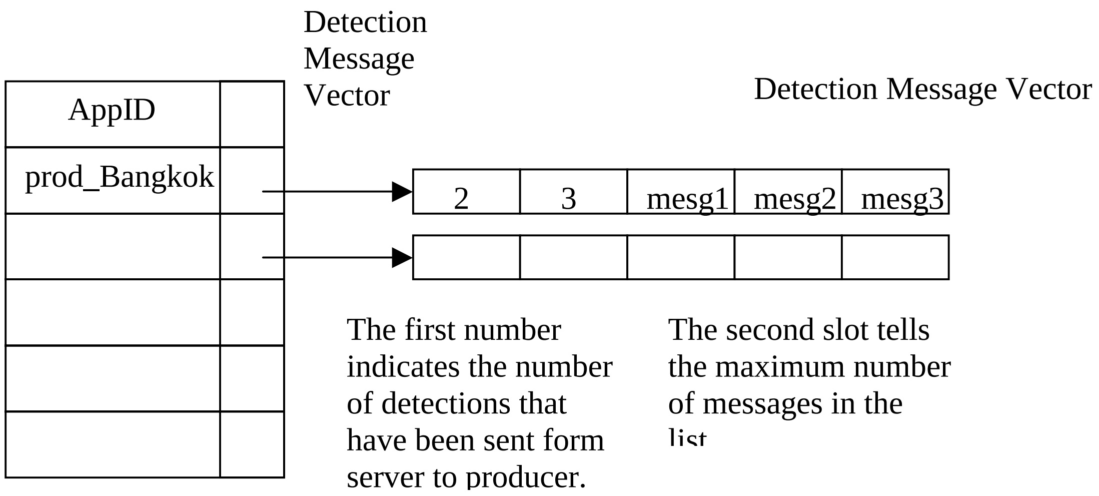

Detection Message Vector
Detection Message Vector
AppID
prod_Bangkok
2
3
mesg1
mesg2
mesg3
The first number indicates the number of detections that have been sent form server to producer.
The second slot tells the maximum number of messages in the list

Figure 6-1 Producer Event List

Recovery: Recovery process needs to rebuild the detection request list of each producer. The key value here, producer id, is extracted from the detection request message read from the file. If the producer id read from the message already exists in the hashtable it just appends the new message at the end of the Vector associated with this producer. If producer id read doesn’t exist then, recovery process creates a new Vector, the first two elements of which are integers. Messages for the new producer are appended to this Vector from here after. The recovery process is as follows:

1. Instantiate prod_DectectnReqstHt hashtable.
2. De-serialize the detectionRequestMessage Objects in sequence from the log file.
3. Get the ProdAppName stored in the detectionRequestMessage object.
4. Invoke put (ProdAppName, DetectionReqstMesg) method on hashtable.

6.1.3 ConsumerList (Hashtable glbEvntName_consumerList)

As shown in Figure 6-2, the consumer list data structure helps the server keep track of the subscribers of each event. The server uses the event name to search for the applications that has subscribed to this event. The hash table maps event name with the vector of consumer Ids.

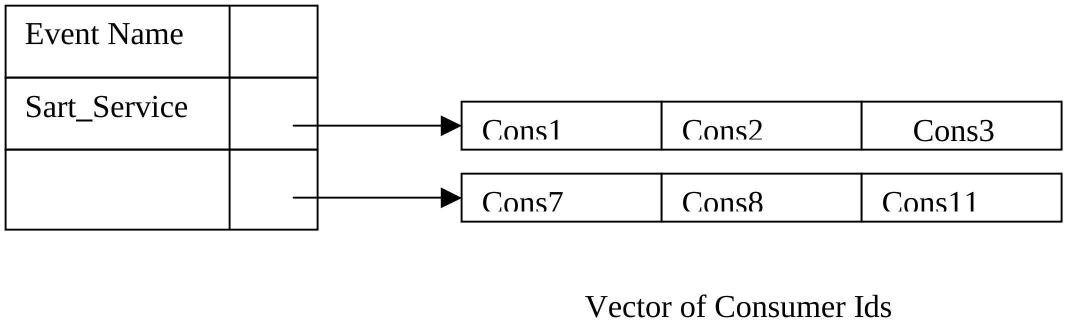


Figure 6-2 Consumer List

Key: EventName

Value: ConsumerList – Vector of ConsId Strings

Logging: filename: glbEvntName_consumerList.log

Keep appending the String ("a / d" + "," + glbEvntNm+","+consID) objects to end of the file.

Reason: As in Client Address List data structure, here both key and value are string objects. The reasoning and the recovery process are similar to Client Address List data structure.

Recovery: The recovery is as follows

1. Instantiate the glbEvntName_consumerList hashtable
2. De-serialize the String ("a / d" + "," + glbEvntNm+","+consID) Objects in sequence
3. Parse it and get Global Event Name and the consumer Id
4. Get the consumer list corresponding to the global event name and invoke add or remove (consId) method on consumer list Vector.

6.2 GlobalNodeManager

GlobalNodeManager maintains the global event graph in two hashtable data structures. It provides mechanism to access the event nodes and event handles based on their names. Recovery of GlobalNodeManager means the recovery of global event graph.

Attributes:

- Hashtable glbEvntNm_GlbEvntNd
- Hashtable glbEvntNm_GlbEvntHndle

6.2.1 Hashtable glbEvntNm_GlbEvntNd

This hashtable maps the global event names with the global event nodes. Whenever there is a request for new global event, primitive or composite, an event node is created on the server and it is added to this table. This table represents the event graph on the server.

Key: String Event Name

Value: Event

Logging: filename: glbEvntNm_GlbEvntNd.log

Whenever a new event is added, update the in-memory hashtable and write it to file by serializing the entire hashtable as one object. This is one place where the write ahead logging and append mode of logging are foregone for simplicity in implementation. The rationale for this decision is explained in chapter 4 and is summarized below.

Rationale: GED could have written just the Event Nodes to log and populate the hashtable by reading them as done for the other hashtables. The reason for persisting the entire table is that, this table represents the global event graph. When a new node is added to this graph (a new event to the table), if it is a global composite event consisting of global primitive or composite events, it updates data in additional nodes corresponding to the constituent events in memory and GED wants this to be reflected on secondary storage. Depending on the level at which the event is being added, number of internal nodes that are modified varies. This could be as small as two primitive event nodes to entire event graph. This requires GED to log all the event nodes that are updated and keep track of the sequence of these updates to re-do them when recovering. This complicates the logging and recovery process. At this point of time, we want to keep our recovery plan simple so we just write the updated in-memory version of hash table into a file. This is one place where we are not doing WAL.

Recovery: De-serialize the hashtable object from the byte stream in file. For all global composite events nodes of type AND, NOT, SEQ, OR set the commInterface object in them to the current GEDInterface. The commInterface (GEDInterface for global composite) reference present in the event nodes is not persisted and recovered here since the GEDInterface object is recovered independent of event nodes.

6.2.2 Hashtable glbEvntNm_GlbEvntHndle

This data structure maps the event names to event handles. Event handles are used in creating the composite events.

Key: String Event Name

Value: Event Handle Object

Logging: Serialize the Event handle objects to file in append mode. The underlying idea is same; recovery mechanism has to reduce the amount serialization done. As in earlier cases, the key in here (Event Name) is a part of value Object (Event Handle). Hence, it can be retrieved from the event handle object. In an Event handle object the constituent Event node object is made transient. As we recover all the event nodes in another hashtable, we can use the same nodes and rebuild the event handles except for the other handle specific data like producer application name etc. This reduces the amount serialization we do considerably.

Recovery: Recovery process is as follows.

1. De-Serialize the Event Handle objects from the file. Retrieve the event name from it.
2. Get the corresponding event node from eventNm_EventNd hash table.
3. Set the event node in current event handle to this and invoke put (event name, event handle) method.

6.3 BufferManager

Buffer manager takes care of all the messages on the server. It adds the incoming messages to the object store according to the availability of space and dispatches the messages in object store to their corresponding consumers in FIFO order. NotifMesgDispatchThread dispatches message objects from the object store to consumers. If logging is activated, the PullMesgThread takes care of pulling those events from log file that could not be stored in main memory buffer when they arrived onto the server. esnCounter is the count of number of event notification messages that arrived onto the server. This counter is maintained to assign the unique event sequence number to each message that arrives on to GED.

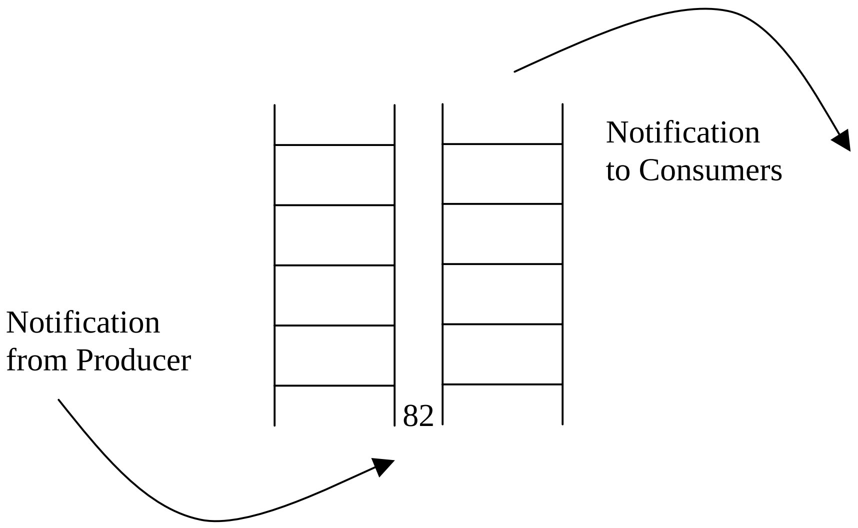
Attributes:

- NotifMesgDispatchThread
- PullMesgThread
- LinkedList objectStore1
- LinkedList objectStore2
- Hashtable clientId_logEvntCounter
- int esnCounter

NotifMesgDispatchThread and PullMesgThread will be re-instantiated and started at the time of recovery. esnCounter is inferred from the maximum sequence number in notification message log file (GED_Notif.log) that stores all the notification messages that arrive onto the server. Recovery of object store is explained below.

6.3.1 Main Memory Buffers (objectStore1 and objectStore2)

This is a queue data structure that stores the notification messages and allows



them to be dispatched in FIFO order.

Figure 6-3 Message Queue

Value: Message Object →{ESN, NotificationMessage}

Logging: There is no specific log file for this queue. It is reconstructed from the notification message log file. Global_Notif.log file contains the Message objects that have arrived onto the server. Along with the indexing information, it also stores the information that indicates the messages that were present in the object store prior to crash. When a notification message from a producer arrives at GED, following is the sequence of steps taken by it:

1. GEDInterface receives the messages through GEDMesgRecvImp interface.
2. This notification message is forwarded to buffer manager.
3. If (Persist) Buffer Manager writes this event into global notification log.
4. If it can be added to Object Store and if (Persist) the buffESN is updated in each log. Message is added to the Object Store
5. If Object Store is full and if (Persist), pendingEvts counter is updated in the log to reflect this information or a warning message is given to the user about the loss of the message.

Recovery: In recovery process the notification log file is read. buffESN read from the log indicates the maximum ESN message that is present in object store. dESN indicates the maximum ESN message that is dispatched to its consumers. These variables convey that the buffer before crash contained the messages with ESN greater than dESN and less than or equal to buffESN. Accordingly, all the events from the event with (ESN > dESN) to the event with (ESN = buffESN) are pulled from the log to main memory and stored in the appropriate object store according to their ESN. The messages with odd ESN are store objectstore1 and messages with even ESN are stored in objectstore2. This restores the object store. The maximum ESN present in the log indicates the esnCounter.

6.3.2 Slow Consumer buffers (Hashtable clientId_logEvntCounter)

This hashtable maps the clientId’s with a message list associated to that client. This message list stores the count and messages that were not picked up by the consumers. This might be because the consumer is slow or it is down. GED demarcates this with a limit on the size of the list associated with each slow consumer. If it exceeds the TIMEOUT specified in the configuration file, GED infers the consumer crash.

Key: String clientId

Value: Vector with the count as first element

Logging: Messages associated with each client are stored in separate log files. These log files are differentiated from other log files with their name. Appending clientId with “_crash.log” derives each log file name. The first four bytes of the file store integer value indicating the number of messages in log file. Any new message is appended at the end of the file and the counter information in the first four bytes is updated.

Recovery: For each clientId_crash.log file, first four bytes are read. If this exceeds the TIMEOUT variable, recovery process knows that the client has crashed. Therefore, the first element of the corresponding Vector is set and the key value pair of clientId and Vector is inserted into the hashtable. If the count read from the first four bytes is less than the TIMEOUT variable then, all the messages are read from the log file and added to a Vector in the same order. First element of the Vector is set to store the count.

6.4 Processing Undelivered Messages

Even when the GED is down, client application can still be up and generate new event notification messages. Unaware of GED crash, client applications try to send event notification to server. Messages sent to server when it was down will be lost. To avoid loss of these messages and discrepancies in event detection due to the loss, they should be stored and sent to GED at a later point of time when GED is restored to its state prior to crash. These messages will be logged on client site. As there is an overhead associated with logging the user is given an option to turn off this logging by setting the LOCAL_LOGGING property in application configuration file.

During the recovery process, after restoring the state, GED queries each client for any messages that were logged locally and collects them from the clients. These messages are collected on client basis not in the (global) order of their generation. As mentioned earlier, they have to be sorted on their time of occurrence before processing. To maintain the global order of event occurrences, GED sorts the event notification messages from different clients using a Java Comparator class that takes the time stamp associated with these messages as basis for comparison. The assumption here is that all the client clocks are synchronous. Processing the messages in sorted order ensures that they are processed in the same order they were generated, the only drawback being the delay in processing because of crash.

6.5 Summary

This chapter explains the details of persisting the data and retrieving them from log files during recovery. The entire recovery process can be summarized as follows: On starting the GED in RESUME mode following a crash, it tries to restore its previous state from the log file generated in the previous run. The entire recovery process is locked by the RecoveryLock explained in previous chapter. GED first acquires the recovery lock. Then it restores the state by restoring the GEDInterface, GlobalNodeManager and BufferManager objects. During the process, the remote interfaces used for the communication with clients are re-bound in the RMI registry. It then compresses the notification message log file so that it does not face the log overflow problem early in this fresh run. Finally it queries the client applications for any messages that were generated when it was down, collects them and process them in the order of their generation. It then releases the recovery lock and continues the normal operation.

CHAPTER 7

Sample Scenario

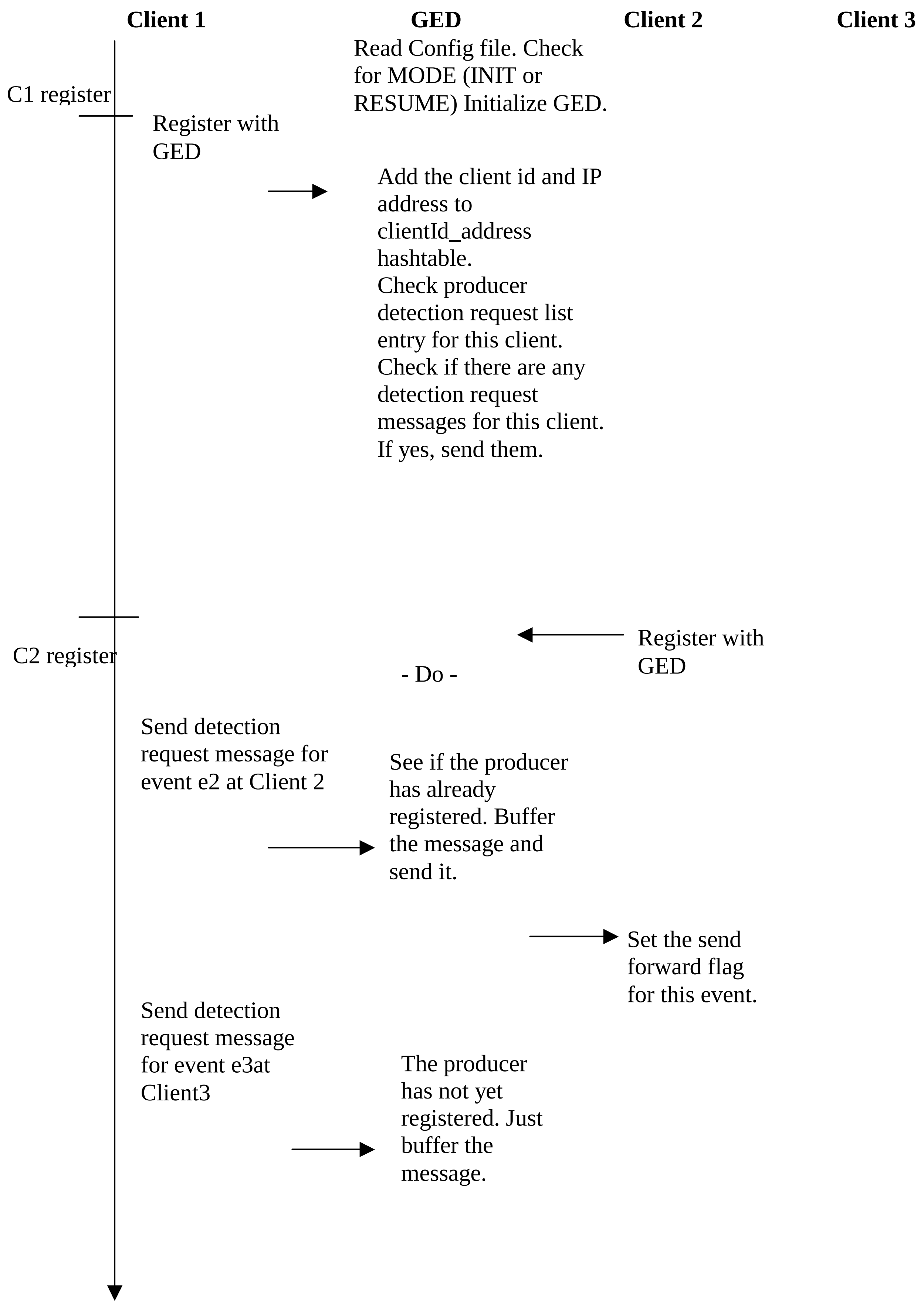
This Chapter shows a sample scenario that exhibits the robustness of GED to system failures. It first shows the normal operation of GED and then the GED operations during crash recovery. Global Configuration file is set as follows:

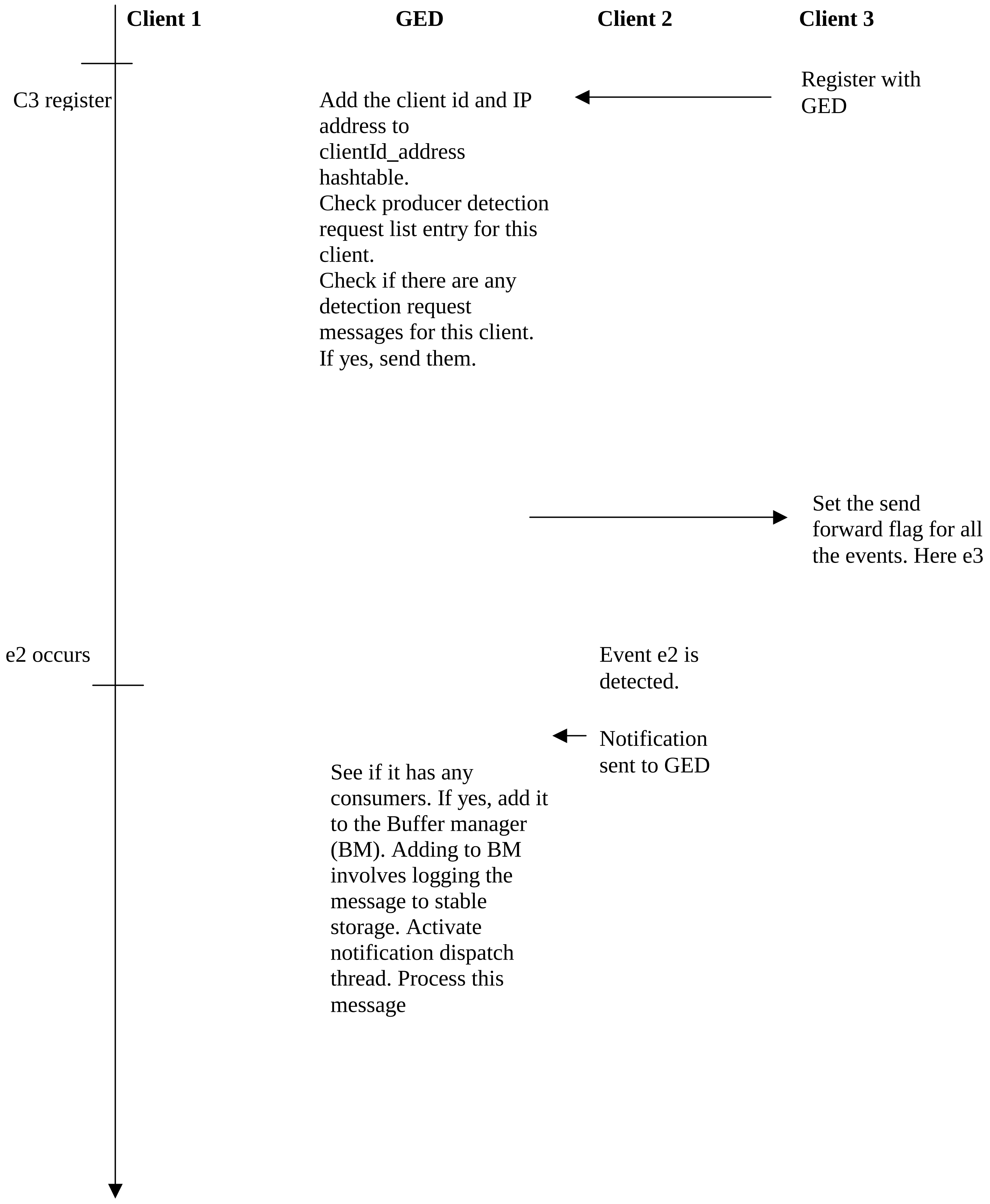
Flag	**Values**
GED_NAME	GED1
MODE	INIT
PERSIST	TRUE
LOG_DIR	To log directory
BUFF_MAX	500
LOG_SIZE	5000
TIMEOUT	5
MAPPING	String Values

Table 7-1 Global Configuration File for this execution

Initially GED is started in INIT mode of operation. The PERSIST property is set to true to enable the recovery of GED in case of system failure. BUFF_MAX, LOG_SIZE, and TIMEOUT variables are set to values shown in the Table 7-1. LOG_DIR and Mappings are set accordingly.

This run of GED logs all the event information. In case of system crash, GED is started in RESUME mode of operation. GED then tries to recover the previous state from the log files in LOG_DIR. After recovery, it continues to provide normal services as earlier.


Client 1
GED
Client 2
Client 3
Read Config file. Check for MODE (INIT or RESUME) Initialize GED.
C1 register
Register with GED
Add the client id and IP address to clientId_address hashtable. Check producer detection request list entry for this client. Check if there are any detection request messages for this client. If yes, send them.
C2 register
Register with GED
- Do -
Send detection request message for event e2 at Client 2
See if the producer has already registered. Buffer the message and send it.
Set the send forward flag for this event.
Send detection request message for event e3at Client3
The producer has not yet registered. Just buffer the message.



Register with GED

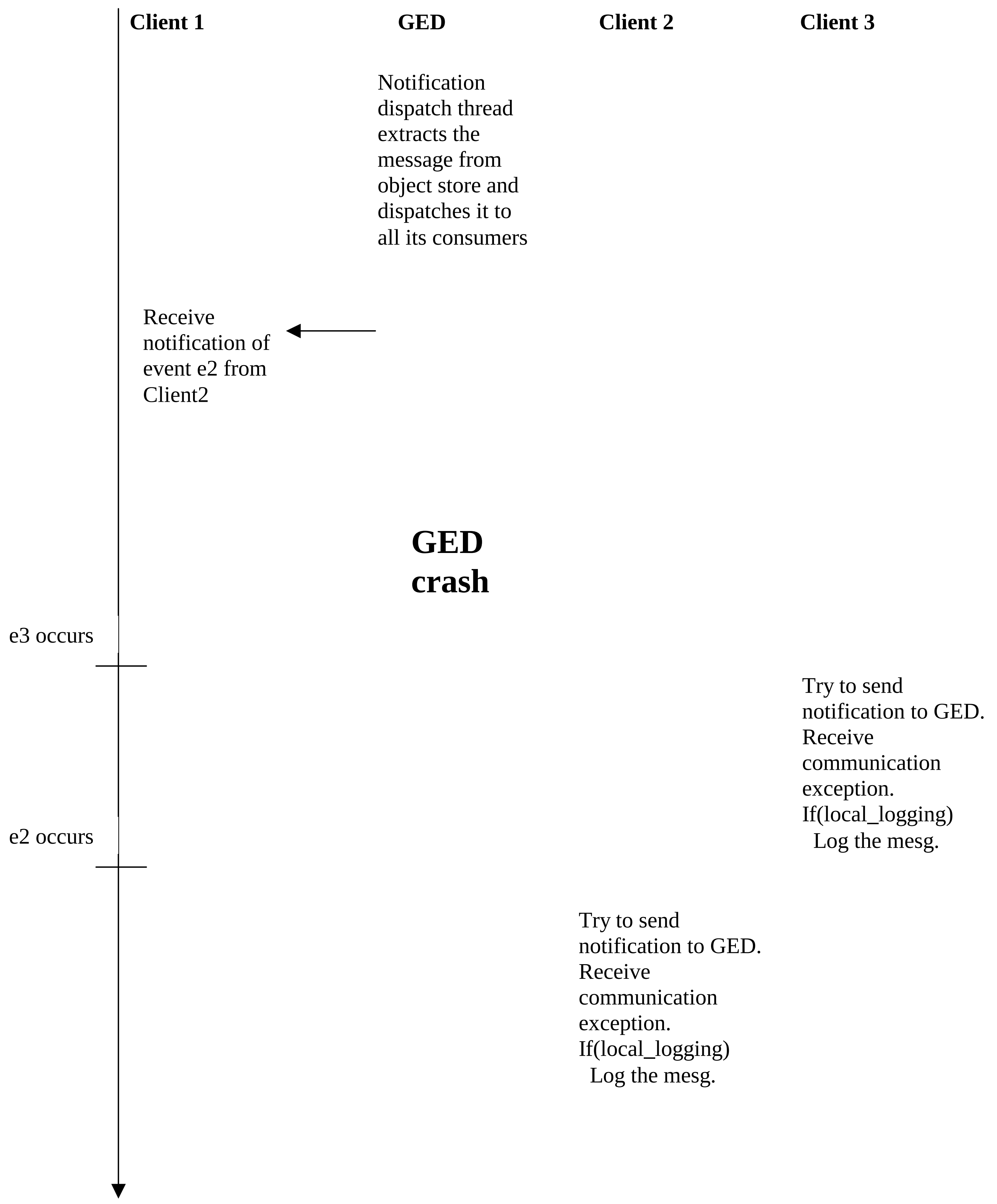
Add the client id and IP address to clientId_address hashtable.
Check producer detection request list entry for this client.
Check if there are any detection request messages for this client.
If yes, send them.

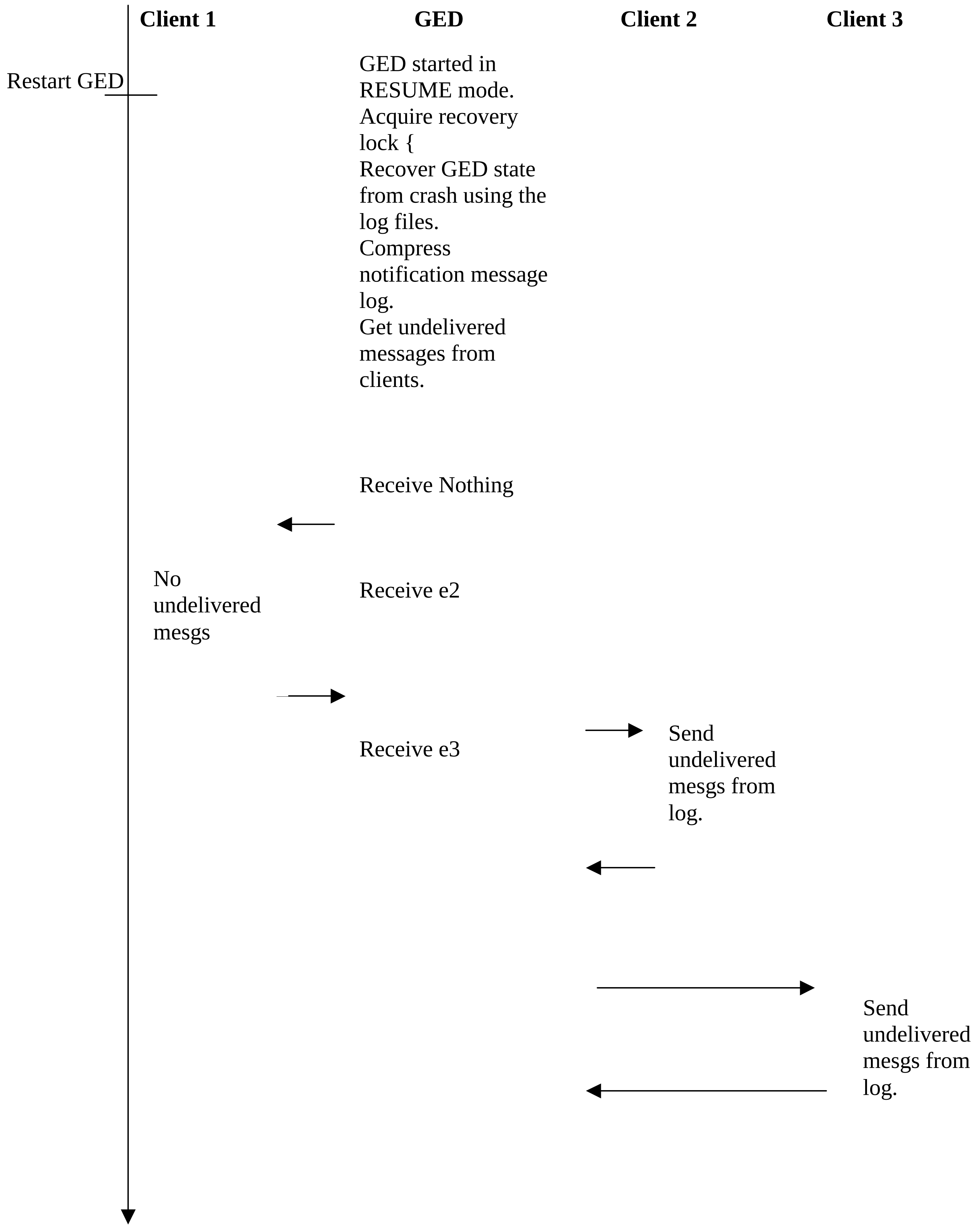
Set the send forward flag for all the events. Here e3

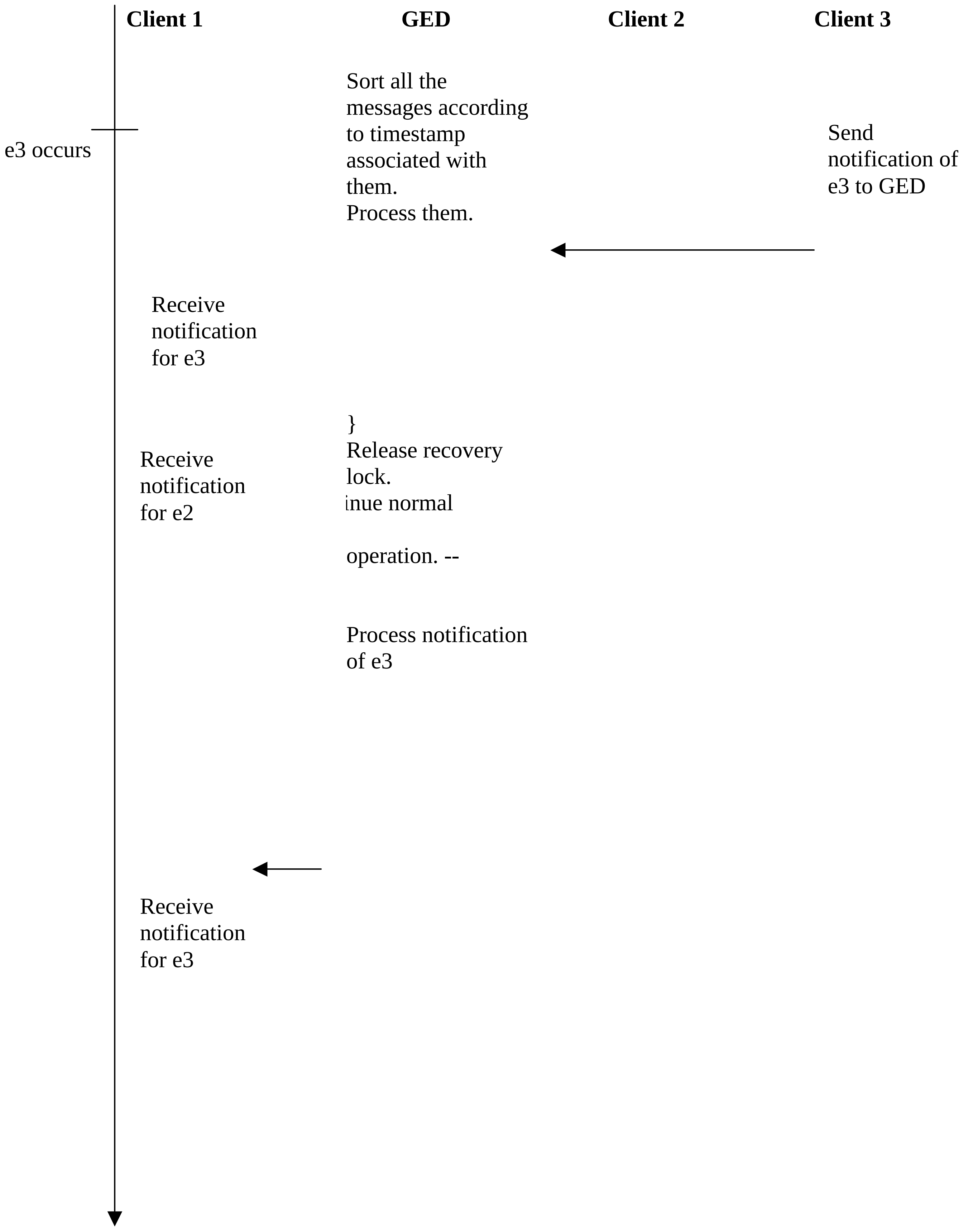
Event e2 is detected.

Notification sent to GED

See if it has any consumers. If yes, add it to the Buffer manager (BM). Adding to BM involves logging the message to stable storage. Activate notification dispatch thread. Process this message


Client 1
GED
Client 2
Client 3
Notification dispatch thread extracts the message from object store and dispatches it to all its consumers
Receive notification of event e2 from Client2
GED crash
e3 occurs
Try to send notification to GED. Receive communication exception. If(local_logging) Log the mesg.
e2 occurs
Try to send notification to GED. Receive communication exception. If(local_logging) Log the mesg.


Client 1
GED
Client 2
Client 3
Restart GED
GED started in RESUME mode.
Acquire recovery lock {
Recover GED state from crash using the log files.
Compress notification message log.
Get undelivered messages from clients.
Receive Nothing
No undelivered mesgs
Receive e2
Receive e3
Send undelivered mesgs from log.
Send undelivered mesgs from log.


Client 1
GED
Client 2
Client 3
Sort all the messages according to timestamp associated with them.
Process them.
e3 occurs
Send notification of e3 to GED
Receive notification for e3
}
Release recovery lock.
inue normal
operation. --
Receive notification for e2
Process notification of e3
Receive notification for e3

CHAPTER 8

Conclusions and Future Work

8.1 Conclusion

This thesis hones the existing system with buffer management, event persistence and recoverable capabilities. The buffer manager module introduced here manages the main memory used to store the notification messages from clients. It also handles the required reads and writes to secondary storage. GED can now be run in either PERSIST or NO-PERSIST mode. In PERSIST mode of operation all the event information and the state of the GED server is persisted on to stable storage. Write Ahead Logging concept is used for this purpose. GED is made robust to system failures. GED can now be recovered to previous consistent state following a system failure and can continue to provide normal services when it recovers. Following a system crash, GED can be recovered by starting it in RESUME mode.

Chapter 1 defines the problem and explains the motivation for this thesis. Chapter 2 reviews the work related to providing persistence and recoverable capabilities. Chapter 3 summarizes the architecture and usage of existing local and global event detector systems. Chapter 4 explains the design issues associated with the buffer manager and providing event persistence and recovery capabilities to GED. Chapter 5 goes into the implementation details of buffer manager and data persistence onto stable storage. Chapter 6 explains the logging and recovery of different data structures required for the GED recovery. Chapter 7 shows an example scenario demonstrating the robustness of GED to system failures and client crashes.

8.2 Future Work

Following could be the extensions to the existing system.

The event detection process now is handled by single thread. It could be extended to a multi threaded event detection process. Then the synchronization of event graph issues should be addressed properly.

Support event monitoring with multiple GEDs over a vast network. A network of GED can be formed to monitor events over a wide network, with each of the GED controlling a subnet and being able share these events with the other GEDs over the network. The replication of GEDs increasing the availability should also be explored.

Collaborate with Distributed Alert Server, which is a recoverable priority based message oriented middleware, used to distribute messages based on publish/subscribe model.

REFERENCES

1. Paton, N.W. *Active database systems*. in *ACM Computing Surveys (CSUR) March 1999*. 1999.
2. Dasari, R., *Events And Rules For JAVA: Design And Implemenation Of A Seamless Approach*, in *Database Systems R&D Center, CIS Department*. 1999, University of Florida: Gainesville.
3. Chakravathy, S. and D. Mishra, *An Event Specification Language (Snoop) for Active Databases and its Detection*. 1991, Database Systems R\&D Center CIS Department University of Florida.
4. Chakravarthy, S. and D. Mishra, *Snoop: An Expressive Event Specification Language for Active Databases.* Data and Knowledge Engineering, 1994. **14**(10): p. 1--26.
5. Tanpisut, W., *Design and Implementation of Event based subscription/notification paradigm for distributed environments*. 2001, The University of Texas at Arlington.
6. Mohan, C., et al., *ARIES: A transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging.* ACM Transactions on Database Systems, 1992. **17**(1): p. 94--162.
7. Rothermel, K. and C. Mohan, *ARIES/NT: A recovery method based on write-ahead logging for nested transactions*, in *Proceedings of the Fifteenth International Conference on Very Large Data Bases*. 1989: Amsterdam. p. 337--346.
8. R.Lorie, *Physical Integrity in a lrage segmented database.* ACM Transactions on Database Systems, 1977.
9. J. Gray, P.M., M. Blasgen, B.Lindsay, R. Lorie, G. Putzolu, T. Price and I. Traiger, *The Recovery Manager of the System R database manager.* ACM Computing Surveys, 1981.
10. SunMicrosystems, *Java Message Service Specification Version 1.0.2b*. 2000.
11. Liao, H., *Global Events in Sentinel: Design and Implementation of a Global Event Detector*, in *MS Thesis*. 1997, Database Systems R&D Center CISE University of Florida, Gainesville, FL 32611.
12. Sung, J.C., *A Recoverable Asynchronous Event Manager for Supporting Distributed Active Databases*, in *E470 CSE Building, Gainesville, FL 32611*. 1997, Database Systems R&D Center CISE University of Florida.
13. Chakravarthy, S., et al., *Composite Events for Active Databases: Semantics, Contexts and Detection*, in *Proc. Int'l. Conf. on Very Large Data Bases VLDB.* 1994: Santiago, Chile. p. 606--617.
14. Mysore Ganesha Rao, Y., *An Agent based approach for extending the Trigger capability of Oracle*, in *ITLAB, CSE department*. 2002, University of Texas at Arlington: Arlington.

15. Subramaniam, N., *A mediator based approach to support ECA rules in DB2 RDBMS*. 2002, The University of Texas at Arlington: Arlington.
16. Gopalakrishnan, G., *Making Sybase fully Active: Supporting Composite events and Prioritized rules*, in *ITLAB, CSE Department*. 2002, University of Texas at Arlington: Arlington.
17. SunMicrosystems, *Object Serialization Specification Sun Microsystems Inc. 2001*. 2000.
18. David Lomet, G.W. *Efficient Transperant Application Recovery In Client-Server Information Systems*. in *SIGMOD*. 1998.
19. Lea, D., *Concurrent Programming in Java*. Second Edition ed. 2000.
20. SunMicrosystems, JavaTM 2 Platform, Standard Edition, v 1.4.0 API Specification. 2002.

BIOGRAPHICAL SKETCH

Sreekant Thirunagari was born on December 21, 1976 in Nizamabad, India. He received his Bachelor of Science degree in Electronics and Communication Engineering from Jawaharlal Nehru Technological University, Hyderabad, India in June 1999. In the Fall of 1999, he started his graduate studies in Computer Science and Engineering at The University of Texas, Arlington. He received his Master of Science in Computer Science from The University of Texas at Arlington, in May 2002. His research interests include active and mobile databases.