Pattern-Based Approach for Logical Traffic Isolation Forensic Modelling

[1]Innocentia Dlamini, [2]Martin Olivier
(ICSA Group) Department of Computer Science
University of Pretoria
Pretoria, RSA
[1]idlamini, [2]molivier{@cs.up.ac.za}

Sihle Sibiya
CCIW, DPSS
Council for Scientific and Industrial Research (CSIR)
Pretoria, RSA
ssibiya@csir.co.za

***Abstract*— The use of design patterns usually changes the approach of software design and makes software development relatively easy. This paper extends work on a forensic model for Logical Traffic Isolation (LTI) based on Differentiated Services (DiffServ) and designs the LTI model using different design patterns. Since design patterns add reliability, flexibility and reusability characteristics to a software system, this paper focuses on three design patterns in modeling the LTI architecture to achieve reusability and flexibility of the LTI model. This model is viewed as a three-tier architecture, which for experimental purposes is composed of the following components: traffic generator, DiffServ network and the sink server. The Mediator pattern is used to coordinate the traffic generator, sink server and, or preservation components, that is DiffServ domain is considered as the mediator. This study uses different design patterns to show how various patterns can be used to design the system.**

Keywords-Decorator Pattern; Observer Pattern; Mediator Pattern; Network forensic; Differentiated services

I. INTRODUCTION

Network Forensics involves the capturing of network traffic by means of scientific and legal procedures that are acceptable in a court of law [1]. This discipline therefore entails the gathering, preserving and analysing of network events in order to discover the source of an attack or other network problem [1] [2] [3]. Network Forensics requires the isolation of malicious network packets [4]. This isolation depends on easy and accurate identification of malicious packets, as well as on forensically sound evidence collection. Strauss et al. [5] proposed a scheme that utilises Differentiated Services (DiffServ) to isolate malicious traffic logically from normal traffic. Since DiffServ is a standard technique, this could well reduce cost. If a DiffServ infrastructure is already in place where an investigation needs to be performed, evidence collection could be facilitated with minimal changes to the network. When the traffic has been identified as malicious, DiffServ uses ingress router as a marking station to mark the detected packet. The marked traffic is furthermore placed in the dedicated queues for quick routing, so as to isolate it from the rest of the traffic.

The DiffServ approach allows Network Forensic investigators to attach both their marking station (ingress router) in isolating the suspicious traffic and preservation station to a cyber victim's network to investigate the case at hand [6]. The advantage of this approach is that it requires minimal network downtime and most importantly minimal network reconfiguration. This DiffServ-based scheme makes provision for a preservation station to store records of the isolated traffic with a view to later analysis. However, in order to minimise network transmission problems such as transmission delays and high network traffic, the preservation station only stores records related to malicious network traffic [6].

We are currently busy implementing the model to test its operation empirically. This requires two support modules (a traffic generator and a sink server) that are not part of the model per se. However, since patterns are also useful for these experiment support modules, they are included in the discussion below. In this paper we use various software design patterns [7] to design the Logical Traffic Isolation (LTI) model based on the DiffServ scheme. The LTI model has three nodes including the support modules, namely; client side that is the users sending data (we use traffic generator for experimental purposes), DiffServ domain, and sink server that is the users receiving data and, or preservation station when there is intrusion that has been detected [6]. The traffic generator that is for experimentation processes is set up in an environment where both malicious traffic and normal traffic are generated. The rest of the paper is structured as follows: Section 2 introduces the LTI model by presenting it as three-tier architecture. Section 3 presents a design of the LTI model by using various design patterns, while Section 4 presents the rule of thumb for the participating design patterns; and Section 5 concludes the paper.

II. THE LOGICAL TRAFFIC ISOLATION MODEL

The LTI model can be divided into three components if we include the two supporting components for experimentation process, namely the traffic generator and the sink server. This therefore results into the model having traffic generator, DiffServ network and the sink server. The traffic generator constitutes the presentation tier, the DiffServ symbolises the logic tier (or application processing tier) and the sink server stands for the data management tier, as in figure 1. These tiers are logically separated from each

other to ensure that a physical change on any of them does not affect all tiers. Figure 1 shows the layout of the LTI model as a three-tier architecture, which is discussed in the subsections that follow.

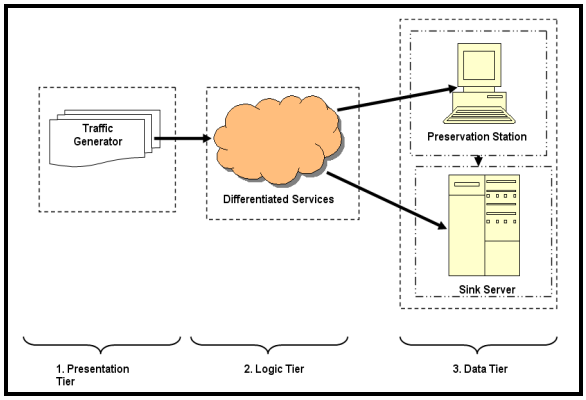

Figure 1. The Logical Traffic Isolation Model

A. *Presentation Tier as Traffic Generator*

This tier allows the user to send and receive data. It is called traffic generator for experimental purposes. In real life, these are the users, in other words legitimate users (users with normal and legal intentions) and/or suspicious users (users with malicious and illegal intentions). These users transmit either normal traffic or suspicious traffic. The traffic generator tier is only a viewpoint for our model; it is not our main concern. Our focus is on the behaviour of the traffic that is generated. The main goal of the LTI model is to isolate the suspicious traffic and to record it by using the preservation station before it is sent on to its destination [5]. The traffic generator tier communicates with the logic tier, composed of DiffServ services tier, during the transmission process. Below is an overview of the Logic tier as a differentiated service (DiffServ).

B. *Differentiated Service as Logic Tier*

The DiffServ services tier controls data flow by performing traffic isolation process. It basically checks the type of traffic it receives from the traffic generator tier, marks it according to its behaviour (either normal or suspicious), routes it through the network and finally unmarks the packets before they leave the DiffServ tier for the data tier. The data tier in the LTI model contains the sink server object. When no suspicious traffic is detected, the sink server is the only object that is active; otherwise, the tier also includes the preservation station. The data tier is discussed in detail in the subsection that follows next.

C. *Preservation Station and Sink Server as Data Tier*

Any transmitted traffic that is found to be suspicious is routed via the preservation station to be recorded before it is permitted to move on to the sink server. The aim of this procedure is to preserve evidence for later analysis, and also to avoid packet loss. Forensic investigators are particularly interested in all suspicious traffic. When the recording is finished, the suspicious traffic is forwarded to the sink server. Normal traffic, on the other hand, is routed directly from the DiffServ services tier to the sink server for processing and storage. When developing the object architecture, it is good to know about the design patterns that are available and that are applicable to the intended system [7]. In the next section we show how various design patterns are used to implement the LTI model.

III. THE LTI COMPONENTS WITH DESIGN PATTERNS

Design patterns add reliability, flexibility and reusability characteristics in a software system [8]. This paper therefore focuses on three design patterns in modelling the LTI architecture. The decorator pattern is used to randomly wrap the behaviour of the generated traffic. The observer pattern [8] [9] is interchangeably used in most of the components of the LTI model, including the traffic generator, DiffServ, preservation station and sink server. The mediator pattern is used to coordinate the traffic generator with the preservation station or the sink server components; the following subsections explain how the different design patterns are applied and class diagrams are provided to show the relationships and interactions.

A. *The Decorator Pattern*

One of the main characteristics of the decorator pattern is to wrap an object to provide a new behaviour [9]. In this work, the decorator pattern is used to randomly wrap any traffic generated with either normal or suspicious behaviour. Figure 2 depicts the use of the traffic generator in a class diagram, which involves five or more classes. The traffic generated can be used on its own and can be considered either as normal traffic or its behaviour can be wrapped to form suspicious traffic.

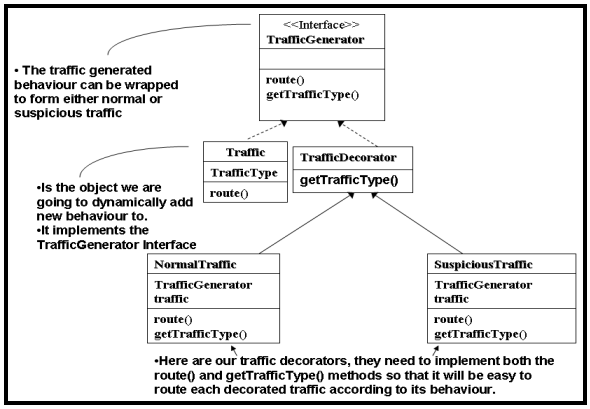

Figure 2. Traffic Generator and Decorator Pattern

The class *Traffic* is the object to which we are going to dynamically add new behaviour; it implements the *TrafficGenerator* interface. Furthermore, the traffic decorators, i.e. *NormalTraffic* and *SuspiciousTraffic* implement both the route() and getTrafficType() methods in order to make it easy to route each decorated traffic according to its behaviour. The traffic generator also acts as a subject of the observer pattern, which is discussed in the following subsection.

B. The Observer Pattern

This pattern allows different objects that are interested in the functionality of the subject-object to subscribe to it, in order to be notified whenever the state of the subject changes [7]. We apply the observer pattern to each element of the LTI model, namely to the traffic generator, DiffServ, preservation station and sink server. Below, each component is discussed in detail. For the observer pattern, the subject and the observer may agree on the way they communicate. Since the observers of this system are related, it will be a good idea to use the pull method rather than a push. A pull method allows the observer to request the information that may be of interest to each observer [1] [9], while in the push method the subject sends all information to all observers. The observer pattern is discussed below in association with some of the LTI model components.

1) Traffic Generator with the Observer Pattern: In Figure 3, the Observable class keeps track of all the observers (ingress, interior and egress routers) and notifies them when the state of traffic generated is not normal, i.e. when it changes to suspicious. The class TrafficGenerator, a subclass of the Observable class, extends Observable class and inherits its methods. All the observers (routers) implement the Observer interface; this provides an interface to the Observable by means of which to communicate with observers when it has to update them [9].

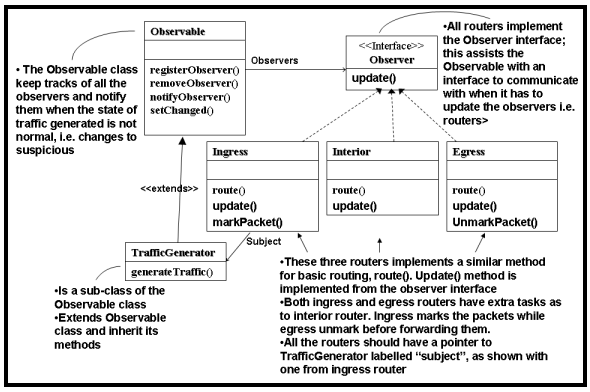

Figure 3. Traffic Generator with Observer pattern

These three observers implement two similar methods for basic routing, namely route() and update(). Both the *Ingress* and the *Egress* routers have an extra task when compared to the *Interior* router. The *Ingress* router marks the packets while the *Egress* router unmarks them before forwarding them to the next node. All the observers (routers) should have a pointer to the *TrafficGenerator* labelled "subject", as is the case with the pointer from ingress router (see Figure 3). The routers, i.e. Ingress, Interior and Egress (from DiffServ domain) act as observers in Figure 3. It can also be a subject when it is related to the preservation station and the sink server. Below, we discuss this option in detail.

2) DiffServ with the Observer Pattern: Figure 4 shows the DiffServ domain acting as a subject and not as an observer as was the case in Figure 3. Both the preservation station and the sink server act as observers. These observers subscribe to the DiffServ to be notified about the state of behaviour of traffic that is currently routed. If the traffic being transmitted is suspicious, DiffServ notifies both observers (see Figure 4). The traffic is then routed via the Preservation station for recording and the preservation of evidence, before it is sent on to the sink server for processing. If the traffic is considered normal, it is sent straight to the sink server. The Observable class keeps tracks of all the observers and notifies them whether the state of traffic that is routed is suspicious or normal. The class DiffServ is a subclass of the Observable class; it extends the Observable class and thus inherits its methods.

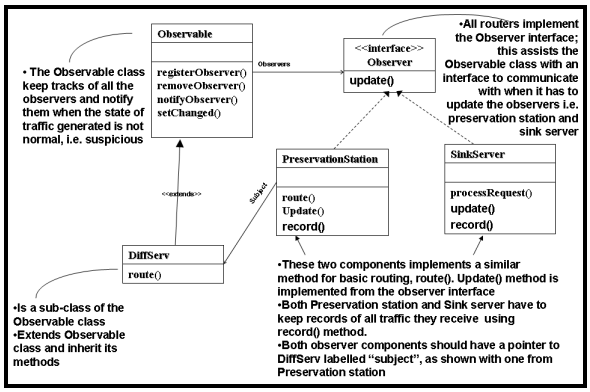

Figure 4. DiffServ with Observer Pattern

All the observers (*PreservationStation* and *SinkServer*) implement the *Observer* interface (Figure 5). This provides the Observable class with an interface by means of which to communicate with observers when it has to update them. Most of the methods implemented by these two observers are similar. This includes – but is not limited to – the route() and update() methods from the Observer interface. Both the preservation station and the sink server keep track of records of all traffic they receive by using the record() method. These observer objects should have a pointer to DiffServ labelled "subject", as is the case with the pointer from the preservation station (see Figure 4). The preservation station

can also act as the subject of the sink server, as shown in Figure 5. This likelihood is discussed in detail in the next subsection.

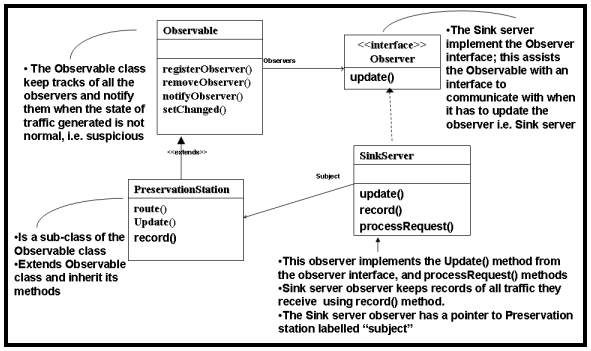

Figure 5. Preservation Station with the Observer Pattern

3) Preservation Station with the Observer pattern: The preservation station as a subject also functions similarly to the traffic generator and DiffServ objects when acting as subjects in the observer pattern, but involves a few minor changes concerning the classes. Figure 5 shows the relationship between the preservation station as a subject and the sink server as an observer object. The subclass of the Observable object is the PreservationStation, with SinkServer as the observer object. This observer implements the Update() method from the Observer interface, and employs processRequest() methods. The SinkServer observer keeps record of all traffic it receives by using the record() method. It also has a pointer to PreservationStation labelled "subject". Figure 6 integrates the three design patterns applied above to form the LTI system. Some Object-Oriented design principles [9] include the following:

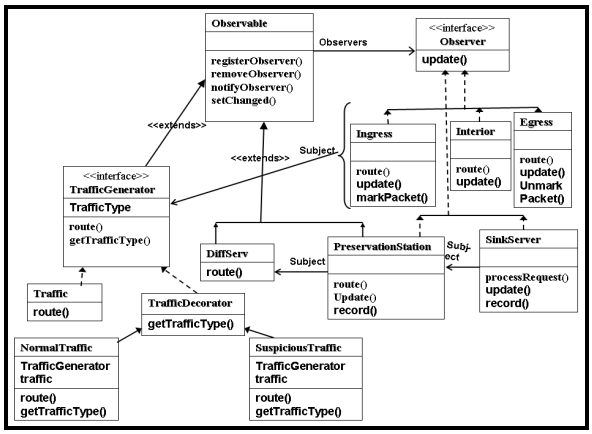

Figure 6. Integration of design patterns applied to the LTI model

"... strive for loosely coupled design between objects that interact (p. 53) …open-close principles (p. 86) … favour composition over inheritance (p. 75)" [9]. The relationship between the subject and the observer in the observer pattern complies with the design principle for favouring composition over inheritance; while the communication between the subject and the observers is kept loosely coupled. The mediator pattern also allows its objects to be loosely coupled. The open-close principle is implemented by the decorator pattern through allowing the behaviour of the traffic generated to be extended without any modification to the entire code. The client and server objects use the DiffServ object for communication. This reduces the number of messages sent between the objects in the system. DiffServ therefore acts as a mediator. The following section contains a discussion of the DiffServ with the mediator pattern.

C. The DiffServ with the Mediator Pattern

The mediator pattern defines an object (i.e. the *DiffServMediator*) that controls the way in which a set of objects interact [7] [10] [11]. All the other classes are completely decoupled from each other. This is achieved by using colleague object to communicate with the mediator (*DiffServ*), rather than having these objects communicating with each other, which results in too much messages transmission among the objects.

In Figure 7, the class *DiffServMediator* simplifies the communication between the *TrafficGenerator* classes, the *PreservationStation* and the *SinkServer*, by implementing the *Mediator* interface. These colleague objects all notify the *DiffServMediator* object whenever their status changes [10].

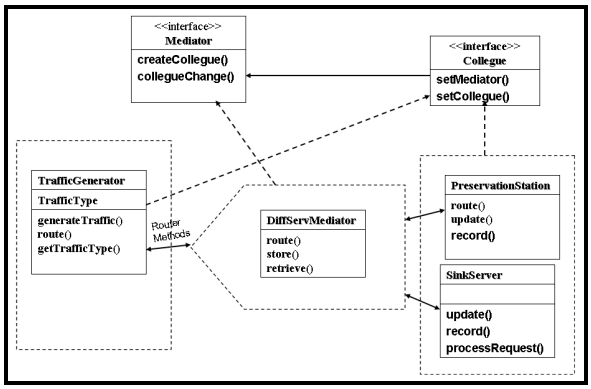

Figure 7. DiffServ as a Mediator

Furthermore, they implement the *Colleague* interface and use the *DiffServMediator* to communicate with each other. The *DiffServMediator* object assists in keeping all objects of the LTI system completely decoupled, which means it complies with the design principle "strive for loosely coupled design between objects that interact" [9]. The *DiffServMediator* contains the control logic (as discussed in Figure 1) of the entire system, when a new object or any necessary logic needs to be added to the system, *DiffServMediator* is used to achieve this. [7]. The relationship between the *Mediato*r interface and the

Colleague interface is the 'many to many' relationships. The section that follows next covers some general principles that can be used to measure the design patterns discussed above.

IV. Rules of thumb for the Used Design Patterns in the LTI Model

Some general guidelines and reckoning for the use of software design patterns that are given below were suggested by the Gang-of-Four [7] and the Head First team [9]. These principles are based on the experience or common knowledge gathered by these groups. Some rules about the Decorator, Observer and Mediator patterns given by the Gang of Four [7] are as follows:

- Mediator and Observer are competing patterns. The difference between them is that an Observer distributes communication by introducing "observer" and "subject" objects, whereas a Mediator object encapsulates the communication between other objects. We have found it easier to make reusable Observers and Subjects than to make reusable Mediators [GoF, p. 346].
- On the other hand, Mediator can leverage Observer for dynamically registering colleagues and communicating with them [GoF, p. 282].
- Mediator gets senders and receivers to reference each other indirectly. Observer defines a very decoupled interface that allows for multiple receivers to be configured at run-time [GoF, p. 347].

Various other design patterns can also be applicable in designing the LTI model, for example the Acceptor and Connector pattern [12] can be used in between the components; the MVC can be used instead of the Observer- in that way the controller will allow the analyst to select traffic to observe. The design patterns used in this study are the most relevant to this work. This increases the reliability of the software system, which in turn reduces development complexity.

V. Conclusion

The work in hand demonstrates the use of various design patterns for constructing the LTI system so as to end up with a more reliable, reusable, convenient and less complex system. The designed system is hope to capture and preserve the detected data to its best. We are currently busy with the implementation of this designed work. In future, the attention in our research work could well be focused on the bidirected LTI model in DiffServ networks.

VI. References

[1] Corey, V., Peterman, C., Shearin S., Greenberg, M.S. & Van Bokkelen, J. 2002, Network Forensics Analysis, Internet Computing, Volume 6, pp. 60- 66, IEEE.

[2] Solomon, M.G., Barrett, D. & Broom, N. 2005, The Need for Computer Forensics, in L. Newman and W.G. Kruse (Eds), Computer Forensics Jump Start, pp. 01-20, SYBEX inc.

[3] Kohn, M., Eloff J. & Olivier, M.S. 2006, Framework for a Digital Forensic Investigation, in H.S. Venter, J.H.P. Eloff, L. Labuschagne and M.M. Eloff (Eds), Proceedings of the ISSA 2006 from Insight to Foresight Conference, Sandton, South Africa (published electronically).

[4] Zantyko, K. 2007, Commentary: Defining Digital Forensics, Forensic Magazine, 20, Vicon Publishing, Feb-March 2007 issue, [Online] Available at: http://www.forensicmag.com/articles.asp?pid=130, as on 12 April 2008.

[5] Strauss, T., Olivier, M.S. & Kourie, D.G. 2006, Differentiated Services for Logical Traffic Isolation, in M.S. Olivier and S. Shenoi (Eds), Advances in Digital Forensics II, pp. 229-237, Springer.

[6] Dlamini, I., Olivier, M.S. & Grobler, M. 2009, A Simulation of Logical Traffic Isolation Using Differentiated Services, *Digital Forensics & Incident Analysis (WDFIA 2009) unpublished.*

[7] [GoF] Gamma, E., Helm, R., Johnson, R. and Vlissides, J. 1996, Design Patterns. Elements of Reusable Object-Oriented Software. Addison-Wesley. ISBN 0-201-63361-2.

[8] Shalloway, A. & Trott, J. 2001, Design Patterns Explained: A New Perspective on Object-Oriented Design, Addison-Wesley.

[9] Freeman, E. & Sierra, K. 2004, Head First Design Patterns, Volume 1, O'Reilly Media, Sebastopol (CA), USA.

[10] Bains, K. & Lau, E. 2002, Mediator Design Pattern. Available at: http://sern.ucalgary.ca/courses/SENG/443/W02/assignments/Mediator/, University of Calgary.

[11] Black, S. 2004, Mediator Design Pattern. http://stevenblack.com/PTN-Mediator.ASP. Steven Black Consulting.

[12] Schmidt, D.C. 1997, Acceptor and Connector: Design Patterns for Initializing Communication Services. The 1st European Pattern Languages of Programming Conference (Washington University technical report #WUCS-97-07).