NEPnet: A Scalable Monitoring System for Anomaly Detection of Network Service

Sujun Cheng, Zhendong Cheng, Zhongzhi Luan, Depei Qian
Sino-German Joint Software Institute, Beijing Key Laboratory of Computer Network
Beihang University
Beijing, China
sujun.cheng@jsi.buaa.edu.cn

***Abstract*—Anomaly detection is very important for modern network service. Yet it is still a big challenge to conduct effective anomaly detection due to the high rate of service data and the complex correlations among them. Owing to the powerful query language and performance potential, complex event processing (CEP) is very suitable for this situation. In this paper, we present NEPnet, a high-performance and scalable monitoring system, which can process events for anomaly detection of network service in real time. NEPnet is based on CEP and provides a SQL-like language supporting various event correlations. On accepting user-defined queries as input, NEPnet builds a tree-based monitoring net for detailed anomaly detection. Considering the anomaly features of network service, the monitoring net utilizes limit trigger, predicate index and route table for different types of processing nodes in it. Our preliminary experiment results show that NEPnet can effectively detect anomaly of network service, with a high-speed of 100,000 events per second and 3~6 times faster than Esper, a general CEP engine.**

Keywords-complex event processing; network service; anomaly detection; monitoring net

I. INTRODUCTION

Failures in network service can cause transmission delay, data loss and even breakdown of network-aware services, which may lead to great damage. These failures are caused by many reasons, such as software error, overload of database, intrusion, DNS hijacking, etc. A typical example is the breakdown of Google's Gmail on 24 Feb 2009, the overload of data center caused by software error led to an interruption of Gmail service for several hours and greatly affected users.

Using monitoring tools to real-time monitor events from network service for anomaly detection is an efficient way to prevent those failures. However, anomaly detection of network service is a complex task. On one hand, modern network service generates massive data at a high rate, which makes traditional monitoring methods hard to meet the demands. For example, Twitter currently generates 10TB a day, which corresponds to roughly 80MB/sec, supporting 1,200 tweets/sec and 100,000 API calls/sec. On the other hand, the correlations among the data generated by services are becoming more complex. Recent work shows that the failure of various network service is often not independent of each other. A new approach is required for managing the high rate and complex data of network service.

Recently, Complex Event Processing (CEP) systems have increasingly become a choice for applications which require high-volume, low-latency and complex event correlation detection, such as those in stock trading, RFID network processing, Web commerce traffic analysis and more. Owing to its powerful and expressive query language and performance potential [9], CEP is very suitable for anomaly detection of network service by finding patterns matching signatures of known anomalies. For example, it can detect a specific sequence of malicious activities caused by intrusion and give its detailed trace. However, most of the existing CEP systems are devoted to general [5][9][10], without considering the anomaly features of network service.

Finding signatures of anomalies requires an efficient pattern matching model for CEP engine. Though Non-deterministic finite automata (NFA) [5] is the most commonly used method in CEP, it has limitation like processing negation pattern(events that do not occur). Some other research of CEP uses tree model for pattern matching, which is expressive and scalable. ZStream [9] constructs a tree-based plan for both the logical and physical representation of a query pattern and gives a cost model to estimate the computation costs of a plan. To some extent, decision tree [1][7][8], also a popular method for anomaly detection, is similar to tree model. Kruegel [7] applied the ID3 algorithm based on decision-making tree to IDS and effectively improved the speed of intrusion detection; Byungchul [1] employed C4.5 algorithm to construct decision trees from training data. Several approaches have also been proposed in the research field to signature detection. In [11], Morin and Debar applied *chronicles formalism* to correlate alarms issued by intrusion detection analyzers. Compared with them, we consider more about sharing intermediate results between different signatures (queries).

In this paper, we propose NEPnet, a CEP-based monitoring system for anomaly detection of network service, focusing on high processing speed and powerful description ability for event correlations. NEPnet provides a flexible and expressive SQL-like language (Section 3) or network services to describe complex anomaly features, supporting various temporal relationships, partitioning operator and aggregate functions. As discussed above, tree is a good choice of the basic pattern match model to find patterns of anomalies. Based on it, we design a monitoring net (Section 4) to represent user-defined queries to process network events. Though the net-structure

This paper is supported by the 863 project of China under the grant No. 2010AA012404, the China International Science and Technology Cooperation Program from the Ministry of Science and Technology of China under the grant No. 2009DFA12110.

seems some closer to Bayesian Networks, NEPnet is signature-based, and that's different with the probability model of Bayesian Networks. Considering the anomaly features of network service, limit trigger, predicate index and route table are utilized in the monitoring net. Finally, in order to test the performance of the monitoring net, we develop NEPnet (Section 2) and compare it with a general CEP engine (Section 5).

II. OVERVIEW OF NEPNET

In this section, we provide a general overview of NEPnet. Fig. 1 illustrates its architecture. The main components are:

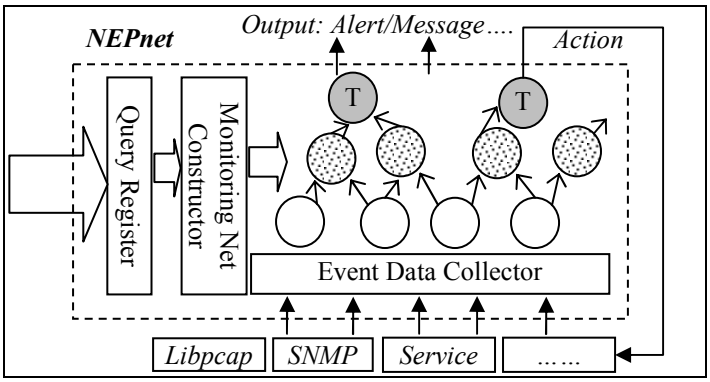


Figure 1. System Architecture of NEPnet

1) Event Data Collector: collect network service data, and sometime may need to encapsulate data into corresponding event types.
2) Query Register: register user-defined queries and related event types.
3) Monitoring Net Constructor: the core of NEPnet . It constructs a monitoring net for the queries registered in Query Register.

During runtime, network and service data flow through the monitoring net, processed in the corresponding nodes of the net.

III. CONTINUOUS QUERY LANGUAGE

One challenge is the complexity of meeting diverse monitor requests in an efficient and scalable way. To fulfill the requirements, similar to [5][9], NEPnet provides a SQL-like continuous query language with various temporal relationships (e.g. *Sequence*, *Kleene*, *Negative*, *Conjunction* and *Disjunction*), partitioning operator (e.g. *Group by*), and aggregate functions (e.g. SUM, MIN, MAX, AVG), making monitoring query more scalable.

Consider one actual anomaly detection for network service. In Windows Server, a series of 529 event & 681 event will be repeatedly trigger if hackers continue trying password attack. 529 event is generated when someone logins to machine via network (with a logon type 3) or via webserver; 681 event is a login failure security event, logged when a user is authenticated from the security authority. Trying many times it could finally trigger 539 event, that is, the account has been locked. It may generate a sequence of events as:

….529,681,529,681,529,681,539,681,539,681….

The detect query on anomaly of network logon can be describe as *Query*1, E529/E681/E539 for short.

Query 1:
SELECT time, E529. *userName*, COUNT (*)
FROM ((E529→E681)[*n*]).WITHIN (*t*)
OR (E529→E681)[1..]→E539 (*userName* =E529. *userName*)
WHERE E529. *userName* =E681.*userName*
AND E529. *logonType* =3

n and t are set to limit frequent and unnecessary alerts, A→B represents event B occurred after event A, A[n].WITHIN(t) denotes event A occurred n times within t . A[1..] means event A occurred at least once.

In addition, another constraints may be added to *Query* 1 : no alert event(AlertE) is generated in 5 minutes. So the temporal constraint turns out to be :

(((E529→E681)[n]).WITHIN (t) OR(E529→E681)[1..]→E539)→!AlertE).WITHIN(5 min)

!AlertE indicates that AlertE event didn't occur.

IV. TREE-BASED MONITORING NET

A. Tree-based Monitoring Net Structure

Tree-based monitoring net is a structure used to filter events with various attributes and relationships. The monitoring net represents all queries.

Fig. 2 shows the hierarchical structure of the monitoring net. Each query is represented by a tree toward the view from the terminal node. Events flow along the directed edges.

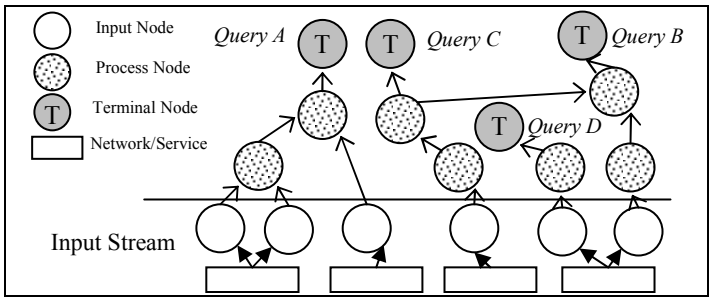


Figure 2. Monitoring Net Structure

There are three node types in the monitoring net:

INode (Input Node): input of the raw network service events, representing different event type, which can come from TCPStream, SNMP agent, Snort, Gmail, etc. This layer plays a role in splitting event streams.

PNode (Process Node): check the constraints of queries, including temporal constraints (e.g. E529→E681, *A* AND *B*, *A* OR *B*, E529 [50] .WITHIN (2 hours)) and predicate constraints (e.g. packet.*port*=135, E529.*username* = E681.*username*). Meanwhile, it also deals with some aggregate operations, such as COUNT, SUM and Group by.

TNode (Terminal Node): indicate an event sequence that meets the query. It provides users with a flexible, scalable interface to define the trigger action.

B. *Detailed Construction*

This section provides some important construction details of the monitoring net, including one method for the basic tree model(Limit Trigger) and two structures for combining queries(Predicate Index && Route Table). In all nodes, a basic approach is to filter unnecessary events as early as possible. So when two event streams meet at a PNode, all predicate constraints related to them are placed in this PNode. When combining queries, predicate constraints are classified into two kinds: single-event and multi-event. Single-event predicate constraint is only concerned with one event type, while multi-event predicate constraint relates to two or more event types. Different structures are designed for these two kinds of constraints.

1) ***Limit Trigger*** *for temporal constraints*: Let us consider the temporal relationship. Fig. 3 shows a monitoring tree built for *Query* 1, which is actually composed of two trees. Take the tree built for temporal constraint ((E529→E681)[*n*]).WITHIN(*t*) as an example. Red edge denotes that when an event passes through it, the parent node will be triggered to process the coming event with the events that have been stored in the right memory of the node, and then send satisfied complex events to its upper node. The event passing through black edge will be only stored in the left memory. Through the limit that only the later coming event (E681) can trigger the processing action of parent node, the temporal requirement E529→E681 (means E681.*timestamp* > E529.*timpstamp*) can be meet . As discussed above, limit the trigger event type can reduce the computer time for checking temporal constraints. In addition, we deal event negation as follow: take A->!B->C as an example, (!B->C) will be combined as a PNode storing the latest *b*, when *c* comes, it sends (*b*,*c*) to upper node, which will iterate the A events stored and find a_n fulfilled *a.timestamp>b.timestamp*, and (b,a_n,c) is a result.

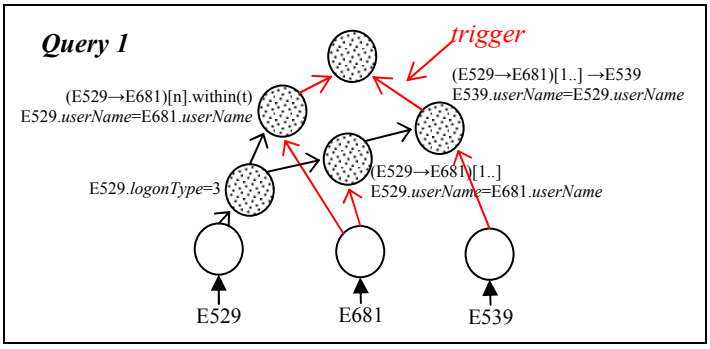


Figure 3. Limit Trigger in the Monitoring Tree for *Query* 1

2) ***Predicate Index*** *for placing single-event predicate constrains*: There are lots of similar single-event predicate constraints in different queries, such as port, group by<*srcIP, destIP, srcPort, destPort*>, average logon times of a network service during 1 hour. For example, port scans often occur at the beginning of an attack, and many queries' conditions contain predicate port checking, such as *port*=21, (21<*port* AND *port*<=135), etc. Much computer time and memory space can be saved if them are combined as Fig. 4. We use the predicate index structure prosed in PSoup[3] for routing. The index tree in Fig. 4 is built with all values from the related queries and plays the role of an index for PNode searching satisfied queries when events come. In addition, Boolean and String value are stored in an array instead predicate index.

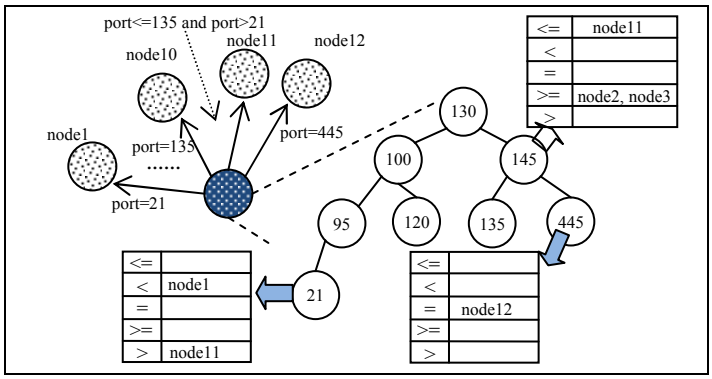


Figure 4. PNode for combining port constraints of many queries

3) ***Route Table*** *for placing multi-event predicate constraints*: A route table is used (as Table I shows) to record the next hop for multi-event. Since NEPnet is a single threaded program and the checking order of the entries in route table determines processing direction, priority is set to each entry in order to improve the processing efficiency. We utilize information entropy[7] to decide precedence. Instead of assigning the same weight to each query, an anomaly weight w_Q is given to each query, related with an anomaly level *al* and the conditional probability P_Q of this query. Users can set the values according to their experience or statistics.

TABLE I. ROUTE TABLE EXAMPLE

Priority	Multi-event Predicate Constraints	Next Hop	Query Set
1	A.a>B.a	node2	Q1,Q3
2	10*A.c <= B.c	node3	Q4
3	A.b>2*B.b AND A.c<B.c	node4	Q6,Q7

Each entry in the route table represents different partitioning ways to query set by using the predicate constraints in it. By partitioning the original set *S*, the information gain of a sub query set will reduce in entropy. The anomaly weight of a query set *S* is calculated by (1) and entropy E_S of *S* is calculated by (2). Then the information gain *G* can be derived by (3) from a given query set S and feature F.

$$W_S = \sum_{i=1}^{|S|} w_{Q_i} \quad (1)$$

$$E_s = \sum_{i=1}^{n} \frac{W_{Q_i}}{W_S} \log_2(W_{Q_i}) \quad (2)$$

$$G_{(S,F)} = E_S - \sum W_{S_V}/W_S * E_{S_V} = \log_2(W_S) - \sum W_{S_V}/W_S * \log_2(W_{S_V}) \quad (3)$$

S_V represents the subsets of *S* divided by feature F, which represent predicate constraints in an entry here. $|S|$ and $|S_V|$ are the number of elements in the query sets *S* and S_V .W_S and W_{Sv} are the anomaly weight of *S* and S_V.

Based on these equations, the table entry with a higher information gain will be set with higher priority, that is, earlier judgment.

In addition, the best way to check this entry is to judge the predicate constraints by the ascending order of the conditional probability of each predicate if there are two or more predicate constraints in an entry (as the entry 3 in Table I).

C. Construction Steps of Monitoring Net

We now turn to the construction steps of NEPnet. Assume that all INodes have been generated when event types get registered with the processing engine.

Step 1: Initialize necessary parameters, such as several w_{Q_i} and P_P for predicate constraints.

Step 2: Do the following for each query:

a) For each single-event constraint: add the constraint to the predicate index of the corresponding INode.

b) For the temporal constraint: combine event types as PNodes from the left. if this PNode type has exist, only need to use the existing one. And set the right event type to trigger the operator of PNode.

c) For each multi-event predicate constraint: find the PNode which includes all related event types, then add the constraint to the proper place in route table by calculating information gain.

Thus far, a monitoring net for the registered queries has been built. Dynamically adding or removing a query is easy for this structure.

V. EVALUATION

We ran a number of experiments to determine the throughput capacity of NEPnet as the network services events load increases, and also test the CPU and Memory utilization. In all experiments, NEPnet was compared with Esper[10], an open source general CEP engine.

1) Throughput Capacity. All experiments were executed on an 8 cores Intel(R) Xen(R) CPU E5405@2.00GHz and 8 GB RAM. System performance was measured by the max processing throughput, which is calculated as follow:

$$throughput = \frac{|Input|}{t_{process}} \quad (4)$$

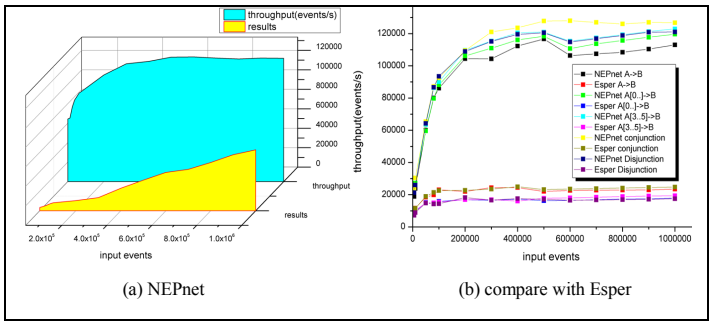
$|Input|$ is the size of input events and $t_{process}$ is the total processing time.

Network service events data were pre-recorded in files and the data was pulled into the system at the maximum rate which the system could accept. For that different query types and input network service data have a significant impact on throughput, a random generator is used to generate input events according the event types provided by Windows Server(e.g. E681, E529, E539, etc.) and queries which contain most event correlations, including temporal constraints (e.g. *Sequence*, *Kleene*, *Negative*, *Conjunction* and *Disjunction*) and predicate constraints. Table II lists the parameters used in the generator.

TABLE II. EXPERIMENT PARAMETERS

Parameter	Meaning	Range
NEvent	number of network service event types	10
NAttr	number of event attributes	13
AttrValue	value range of event attributes	0~100000
NQuery	number of queries (signatures)	30
NTemporal	number of temporal constraints	0~6
NPredicate	number of predicate constraints	0~4

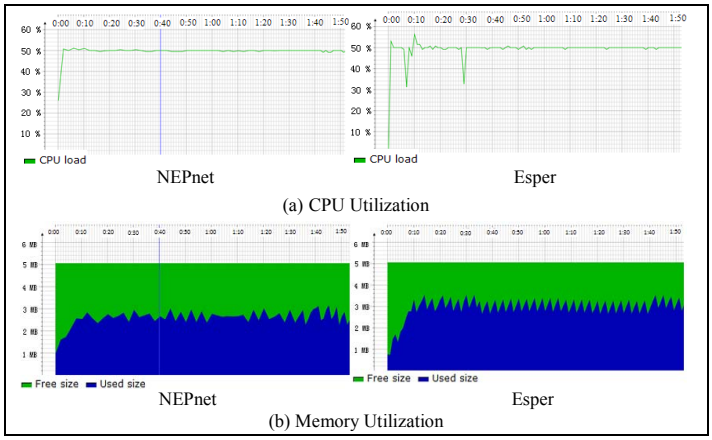
Fig. 5(a) shows the average throughput and the number of detected anomaly results. The highest throughput capacity can reach more than 10,000 events/sec. Observe form Fig. 5(b), our system is 3~6 times faster than Esper.



(a) NEPnet (b) compare with Esper

Figure 5. Throughput

2) CPU and Memory Utilization. The CPU and Memory utilization were evaluated in a dual core processor. As shown in Fig. 6, the CPU utilizations of both NEPnet and Esper are nearly 50%, which denotes that they fully occupied a processor, and NEPnet is a little more stable than Esper. The Memory space used by both of them are less, and NEPnet also has a better performance than Esper.



(a) CPU Utilization

(b) Memory Utilization

Figure 6. CPU and Memory Utilization

VI. CONCLUSION

Failure of network service might cause serious consequences, and early anomaly detection is very important for supporting management. This paper presents NEPnet, a high performance and scalable monitoring system which is designed and implemented to efficiently find signatures of anomalies of network service. The scalable SQL-like language makes NEPnet easily used by network service, supporting various event correlations, such as *Kleene*, *Negative*, *Conjunction*, *Disjunction*, etc. The user-defined queries are represented by a tree-based monitoring net with considering anomaly features of network service. Our preliminary experiments showed that NEPnet efficiently analyzes event correlations and detects anomalies for network service.

REFERENCES

[1] B. Park, Y. Won, H. Yu, J. Hong, H. Noh and J. Lee. Fault Detection in IP-based Process Control Networks using Data Mining. In IEEE/IFIP International Symposium on Integrated Network Management(IM 2009). pp. 211–217, 2009.

[2] S. Chang, X. Qiu. Z. Gao, K. Liu and F. Qi. A flow-based anomaly detection method using sketch and combinations of traffic features. In IEEE International Conference on Network and Service Management(CNSM), pp. 302–305, 2010

[3] S. Chandrasekaran and M. Franklin. PSoup: a system for streaming queries over streaming data. VLDB Journal, 12:140–156, 2003

[4] C. Cranor, T. Johnson, O. Spatschek, and V. Shkapenyuk. Gigascope: A stream database for network applications. In Proceedings of ACM SIGMOD, June 2003 .

[5] E. Wu, Y. Diao, and S. Rizvi. High-performance complex event processing over streams. In Proc. SIGMOD, 2006

[6] T. johnoson, S. Singh, and G. Varghese. On scalable attack detection in the network. In ACM Internet Measurement Conference(IMC), pages 187-200, 2004

[7] C. Kruegel and T. Toth. Using decision trees to improve signature-based intrusion detection. In Proc. Int'l Symp. Recent Advances in Intrusion Detection, 2003.

[8] M. Garofalakis and R. Rastogi. Data Mining Meets Network Management: The NEMESIS Project. In ACM SIGMOD Int'l Workshop on Research Issues in Data Mining and Knowledge Discovery, 2001.

[9] Y. Mei and S. Madden. ZStream: a cost-based query processor for adaptively detecting composite events. In Proc. SIGMOD, 2009.

[10] Esper, http:// http://esper.codehaus.org.

[11] B. Morin, H. Debar. Correlation of Intrusion Symptoms: An Application of Chronicles. Proc. of the 6th International Symposium on Recent Advances in Intrusion Detection, Pittsburgh, PA. USA: Springer-Verleg, 2003.