**EVENT STREAM DIAGNOSTICS**

Author: Yogesh Padmanaban.

# Abstract

While debugging a live site issue, significant amount of time and effort is spent on

1. Isolating the error message in the source code. 2. Determining the pathway of code execution which led to the error. We came up with an approach to make debugging easier.

a. Isolate and retrieve all the observed pathways of execution from logs (Event Streams)

b. Identify the pathways which leads to errors. In addition to making the debugging easier the Event Streams also has some important applicatons Applications. A Service is characterized by its choice of pathways and their frequency of execution. A) We want to see if the lengths of code paths are good enough to determine its health. In a normal day, we observe all possible lengths of Event Streams for that service in a normal day and the percentage of execution of code paths for each length. There is a drastic difference in the lengths of code paths run by service on a normal day vs outage day in our Roaming service. B) We want to catch the timeout errors which are difficult to catch otherwise using the timespan taken by each event stream. C) We want to identify similar Event Streams in each service and their behaviors giving rise to patterns in Event streams structures.

# CONTENTS

Office client interacts with Office Services. Example an Outlook client interacts with Exchange Server [1]. A Sequence of all tasks executed by the server to satisfy the client request is called a server stream. Each of these tasks needs to happen “successfully” to satisfy the client request and are logged on the server side. The beginning, end, and errors of the tasks are logged and identified by an EventID. The status of the task is indicated by a seriousness of the logging level. All tasks occurring on a single client request are tagged with a GUID unique to that request.

Crux of the Idea is as follows:

1. Capture the chronological sequence of eventIDs for a request and give an Identity for it.{EventStream}. Event Stream represents the pathway of Execution for the client request.
2. We analyze pathways diagnostics. A) The most commonly used pathways B) most delayed pathways and C) the most erred pathways.
3. We observe the interaction/relationships between pathways of concurrent requests. We define ‘victim stream’ as ‘affected’ by a ‘neighbor stream’ when the victim stream contains serious error levels when executing along with the “neighbor stream” during in the same time.
4. Cluster the structurally similar pathways. Once clustered we find the relationship between structural similarities and Errors/behavioral characteristics of the pathways.
5. Client request involves performing tasks by several Services along the pathways. We can infer inter-service Relationships within a single pathway by mapping each of the event IDs to their corresponding features or intermediate services run by the request. We can identify individual impact of cycles (service A calls service B & service B calls service A) between services. This in a way gives a notion towards reliability of a service.

We performed the diagnostics on Reading, Roaming and Nexus services using ULS logs obtained from the periods of august 8/31/2013 to 9/02/2013. Nexus is a pipeline for Office client data to flow into cosmos[2]. Cosmos is the Map Reduce data store for storing Microsoft data[3]. Roaming service takes care of cached copies of most recently used data for Office[4]. Reading is an app for reading ebooks[5].

# Unified Logging Service

Unified Logging Service {ULS} represents the Platform for instrumentation/ Logging for Office Server and Client Services[6]. A line of code triggers an event in the ULS log file (a text file), which is then used to debug the service. The following fields are stored in a ULS log line.

1. **Timestamp**: the time of the log line.

2. **Process Name**: The name of the process handling the request. Eg. wsstracing.exe (0x0BDC)

3. **Thread Id**: The thread Id handling the request. Eg. 0x08E0

4. **Area**: The Area the requests belong to. Eg. SharePoint Foundation

5. **Category**: The category, the service belongs to. Eg. Upgrade

6. **Event ID**: identifies the actual code line executed by the Server using a tag. Eg. “fbv7”

7. **Level**: The seriousness of the log line or the impact of the event. It could be one of

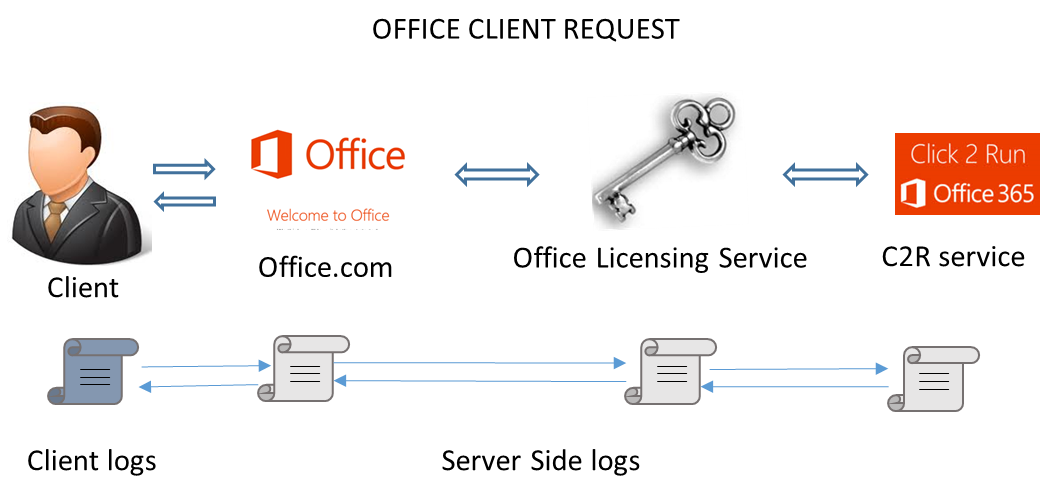
Information, Medium, High, Verbose, Warning, Critical, Unexpected, Exception, Monitorable

8. **Correlation ID**: a GUID which uniquely identifies the individual request from client.

Eg. “b89c8a96-1725-4d55-a95e-dd4fe6223bf5”

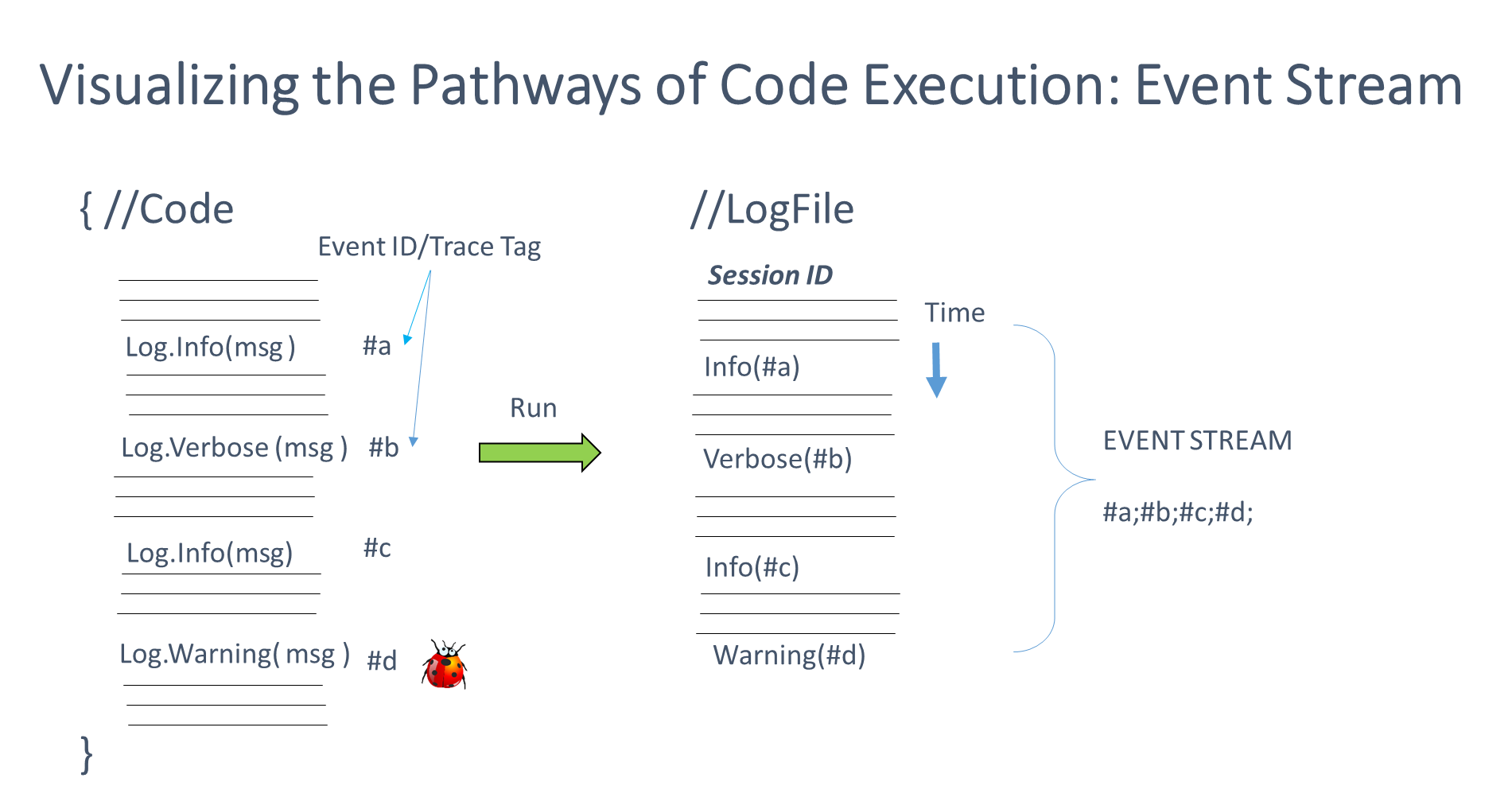
9. **Message**: Contains extra information regarding the logged line.

# Event Stream



**Server Streams**:

A typical client request arises from the desktop/web app and interacts with the hosted services. Services might interact with themselves to satisfy a client request. Services logs their sequence of events during computation on the client request. When the request is completed, client logs an even to record the completion and result of the request. A session ID or correlation ID can be used to track the entire start to end sequence of events including server side events thanks to the logs. For example Click2Run service interacts with Office Licensing Service before answering the clients with their requirements for a requested Office install binary. A Correlation ID can track the entire pathway of client request [7].



**Event Streams**:

There are several simultaneous /concurrently running requests in a server. Each of them throws log line (event) for every step (task) involved in processing a request. Each service records its events in the ULS log. The first step is to isolate the log writes for a single request using the correlation ID or a session ID as given in the picture. If we capture the set of all log lines for a single Correlation ID, we get a stream of Event IDs or a chronological sequence of code lines executed by the Server in order to satisfy the request from the Client. Thus we capture the Server execution for every request from the client. It is termed as **Server Stream**. Event ID identifies the actual code line executed by the Service in response to the Client request. If we isolate only the events id from the log lines and make it a single sequence of event IDs, we get an **Event Stream. Thus Event Stream identifies a pattern of client requests.** The Correlation ID changes for every instance of the same request. But skeleton of the request (Event stream) remains same for particular type of request. We capture all the possible observed event streams from the ULS log for a service. From them we study general diagnostics.

A sample EventStream from Storytelling service would look like the following: The event IDs are separated by means of a “;”.

77a3;xmnv;ala2r;aei4c;aei4d;ahmsa;anhgw;an8kk;xmnv;xmnv;xmnv;xmnv;anhgw;an8kl;anhgw;an8km;aigx1;b1wj;i11i;agw4s;agw4w;i11s;hpk2;cg8i;bhn6;hpk2;cg8i;anhgw;an8kq;9zah;ala2u;77a3;f7zo;anhgw;an8ko

# General Diagnostics

Now that we have captured the Server stream and Event stream of the services, we should realize that we can obtain all possible observed executions for service by the server. There are 2407 possible observed event streams for Reading Service; 91 event streams found for Nexus and 352 for Roaming service. We define Err given by the “level” field taken by the values such as “Warning, Critical, Exception, Unexpected, Monitorable” in a server stream. For every client request we extract the event stream, Start time, End time and the level frequencies for each possible levels. While most of the events have “Verbose” or “Medium” as their level, we can look for the interesting ones. The following diagnostics are useful:

1. **MOST ERRED EVENTSTREAMS AND REQUESTS*:***

The requests and the streams which has most number of errors. For the chosen services Nexus, Reading and Roaming, we found the following maximum number of each error level and the corresponding erring requests for the chosen ULS logs.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Services | Warning | Critical | Exception | Unexpected | Monitorable | High |
| Reading | 0 | 0 | 17 | 40 | 353 | 1 |
| Nexus | 0 | 0 | 0 | 15 | 4 | 2 |
| Roaming | 0 | 0 | 0 | 6 | 11 | 1 |

While the above can be obtained without using the EventStreams, the true power of Eventstream is revealed by the following aggregate queries. {The streams which are humongous are cut short to increase clarity}.

W: sum of Warning C: sum of Critical E: sum of Exception U: Sum of Unexpected M: Sum of Monitorable.

***Service***: Reading

***Category***: Most number of sum of Monitorable error levels in their requests.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| EventStream | W | C | E | U | M |
| 77a3;anpsk;anpsm;anpsy;an9ap;ac7lb;an9an;ao1qf;anpsz;anpsz;anpsz;anps1;anpsy;an9ap;ac7lb;an9an;ao1qf;anpsz;anpsz;anpsz;anps1;anpsy;an9ap;ac7lb;an9an;ao1qf;anpsz;anpsz;anpsz;anps0;anpsn;anpsl;77a3 | 0 | 0 | 0 | 0 | 6008 |
| 77a3;gn9u;xmnv;ala2r;aei4c;aei4d;ahmsa;ac7lv;al5ee; … | 0 | 0 | 0 | 0 | 2897 |
| 77a3;gn9u;xmnv;ala2r;aei4c;aei4d;ahmsa;ac7lv;al5ee;  dcw3;apmy9;… | 0 | 0 | 0 | 0 | 2589 |
| 77a3;gn9u;xmnv;ala2r;aei4c;aei4d;ahmsa;ac7lv;al5ee;  dcw3;apmy9;… | 0 | 0 | 0 | 0 | 2395 |
| 77a3;gn9u;xmnv;ala2r;aei4c;aei4d;ahmsa;ac7lv;al5ee;  dcw3;apmy9;… | 0 | 0 | 0 | 0 | 2344 |

***Service***: Reading

***Category***: Most Number of sum of Exception error levels

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| EventStream | W | C | E | U | M |
| 77a3;gn9u;xmnv;ala2r;aei4c;aei4d;ahmsa;ac7lx;al5ee;dcw3;… | 0 | 0 | 138 | 0 | 1289 |
| 77a3;gn9u;xmnv;ala2r;aei4c;aei4d;ahmsa;ac7lx;al5ee;dcw3; … | 0 | 0 | 72 | 0 | 96 |
| 77a3;gn9u;xmnv;ala2r;aei4c;aei4d;ahmsa;ac7lx;al5ee;dcw3; …… | 0 | 0 | 70 | 0 | 91 |
| 77a3;gn9u;xmnv;ala2r;aei4c;aei4d;ahmsa;ac7lx;al5ee;dcw3; … | 0 | 0 | 36 | 0 | 336 |
| 77a3;gn9u;xmnv;ala2r;aei4c;aei4d;ahmsa;ac7lx;al5ee;dcw3; … | 0 | 0 | 33 | 0 | 310 |

***Service***: Reading

***Category***: Most Number of sum of Unexpected error levels

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| EventStream | W | C | E | U | M |
| 77a3;xmnv;ala2r;aei4c;aei4d;ahmsa;al5ee;dcw3;apmy9;... | 0 | 0 | 17 | 40 | 192 |
| 77a3;gn9u;xmnv;ala2r;aei4c;aei4d;ahmsa;ac7lx;al5ee;… | 0 | 0 | 13 | 25 | 158 |
| 77a3;gn9u;xmnv;ala2r;aei4c;aei4d;ahmsa;ac7lx;al5ee;… | 0 | 0 | 13 | 25 | 156 |
| 77a3;gn9u;xmnv;ala2r;aei4c;aei4d;ahmsa;ac7lx;al5ee;… | 0 | 0 | 9 | 11 | 2 |

***Service***: Roaming

***Category***: Most Number of sum of Critical error levels

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| EventStream | W | C | E | U | M |
| 77a3;mr04;mr08;dcw3;dcw3;akhz0;ak31v;ak31w;ak31z;ak31s;a495;mr09;a495;mr0b;dcw3;agh86;mr0f;mr0h;mr0i;agl1b; … | 0 | 1 | 0 | 0 | 1 |

***Service***: Roaming

***Category***: Most Number of sum of Unexpected error levels

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| EventStream | W | C | E | U | M |
| 77a3;mr04;mr08;dcw3;dcw3;ak31v;ak31s;a495;mr09;a495;… | 0 | 0 | 0 | 44 | 0 |
| 77a3;mr04;mr08;dcw3;dcw3;ak31v;ak31s;a495;mr09;a495;… | 0 | 0 | 0 | 8 | 0 |
| 77a3;aeqvg;aeqvs;9w8h;9w8i;aeqvg;aeqvg;aeqvg;aeqvg;… | 0 | 0 | 0 | 6 | 0 |

***Service***: Roaming

***Category***: Most Number of sum of Monitorable error levels

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| EventStream | W | C | E | U | M |
| 77a3;adihl;adihh;adihw;dcw3;adihc;adiho;adiho;adiho;adihq;als3t;77a3 | 0 | 0 | 0 | 0 | 24 |
| ajmfo;ajmfo;ajmfo;ajmfo;ajmfo;ajmfo;ajmfw;ajmfy;ajmfk;… | 0 | 0 | 0 | 0 | 11 |
| 77a3;aebci;af7xc;aiwcp;akhz0;adhof;adhoi;af7xd;77a3 | 0 | 0 | 0 | 0 | 4 |

***Service***: Nexus

***Category***: Most number of sum of Unexpected and Monitorable error levels in their requests.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| EventStream | W | C | E | U | M |
| aebce;af7xc;ajk04;ajk04;ajk04;ajk04;ajk05;ajk04;ajk04;ajk04;ajk04;ajk05;aebci;af7xc;aebcc;ajk04;aebci;af7xc;aebcc;ajk04;aebci;af7xc;aebcc;ajk04;aebci;  af7xc;aebcc;ajk04;ajk05;af7xd | 0 | 0 | 0 | 1065 | 284 |
| aebci;af7xc;aebcc;aebci;af7xc;aebcc;aebci;af7xc;aebcc;aebci;af7xc;aebcc | 0 | 0 | 0 | 0 | 432 |
| ap3l0;aitqq;ap3l3;af6ok;af6oq | 0 | 0 | 0 | 0 | 268 |
| aebce;af7xc;aebb0;aebcb;ahrz8;ahrz8 | 0 | 0 | 0 | 13 | 26 |

1. ***Longest requests and Streams:*** The requests and streams which has most number of event IDs.

Looking at the graphs reveals several important truths. The length of the pathways chosen by the services vary vastly! Nexus is a pipeline which transfer client usage data to cosmos. The amount of variation in the number of tasks needed to satisfy a client is minimal. The number of eventIds logged are less. Thus there is a huge spike at around 6 events. The outliers are at event length of 60. Reading services seems to have a vast spread of length. This is expected as the amount of time spent in reading/using the tablet/desktop/mobile varies person to person. It could be huge amount of idle events. Roaming has pretty good distribution of stream lengths with just two outliers. It would require a developer’s investigation to precisely explain the internal stories of the above graphs.

**SYSTEM HEALTH AND LENGTH OF EVENT STREAMS:**

Important application of the Length of the streams is that the Health of the System can be gauged by looking at the percentage of execution for each of the lengths of the event streams. There is a vast difference between the normal days of execution vs the service outage day (Jan 14,2014 where the authentication of Roaming Service failed). This resulted in great reduction in the length of the event streams executed and we see a peek in the Event Streams of length 2.

1. ***Most Delayed streams and most delayed request amongst an individual stream:*** Given that the start time and EndTime of a stream, we can find the streams which takes maximum amount of time. There are always possibilities for a specific request to take more than a usual time to complete a task. This is a very important and powerful debugging tool.

Looking at the graphs we see similar behavior in terms of the timespans for the request as compared to the length of the event streams. The variation was huge necessitating a log-log graphs for Reading and Roaming Streams. The Timespan of the Event Streams are useful to debug the Timeout Errors which are otherwise difficult of debug.

1. ***Most commonly executed streams (streams with most number of requests)***: This metric marks the “Usage” or coverage of the code paths. These are the most important code paths where a break could shake the system. The most frequently used Reading, Nexus, Roaming Streams are: The count represents the number of times the request executed the pathways of EventStreams. Percentage represents the percent of usage of the EventStream pathways to the observed all possible EventStream pathways.

|  |  |  |  |
| --- | --- | --- | --- |
| S.No | Roaming Eventstream | Count | Percent |
| 1 | 77a3;77a3 | 17145 | 46.144% |
| 2 | 77a3;akhz6;ahv0v;77a3 | 14246 | 38.34% |
| 3 | 77a3;ajmfo;77a3 | 3844 | 10.345% |
| 4 | 77a3;aeqvg;aeqvs;9w8h;9w8i;aeqvg;aeqvg;aeqvg;aeqvg;… | 7845 | 3.77% |
| 5 | 77a3;aeqvg;aeqvs;aeqvu;aeqvv;ajmah;aeqvv;ajmah;aeqvy;  77a3 | 641 | 1.72% |

|  |  |  |  |
| --- | --- | --- | --- |
| S.No | Reading Eventstream | count | Percent |
| 1 | 77a3;77a3 | 41267 | 35.28% |
| 2 | 77a3;akhz6;ahv0v;77a3 | 29853 | 25.49% |
| 3 | 77a3;ajmfo;77a3 | 10901 | 9.3% |
| 4 | 77a3;aeqvg;aeqvs;9w8h;9w8i;aeqvg;aeqvg;aeqvg;aeqcg;… | 3586 | 3.06% |
| 5 | 77a3;aeqvg;aeqvs;ajkyr;aeqvu;aeqvv;ajmah;aeqvy;  77a3 | 1816 | 1.55% |

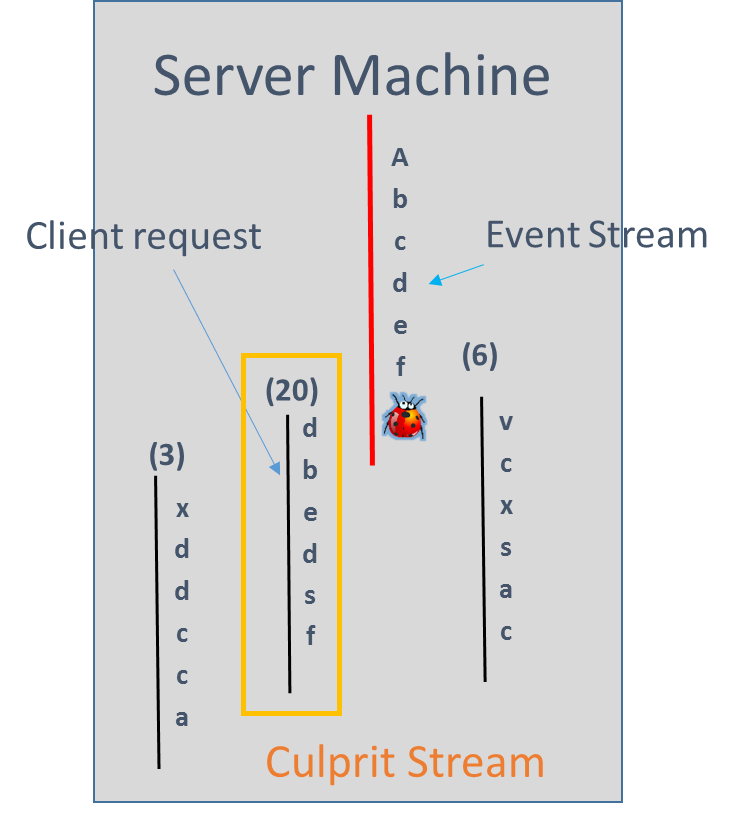
|  |  |  |  |
| --- | --- | --- | --- |
| Sno | Nexus Eventstream | count | Percentage |
| 1 | an8jj;ap3nc;ap3nc;ap3nc;ap3nc;ahv0v | 158703 | 76.39% |
| 2 | apl4f;an3fm;an3fw;apl4i | 15222 | 7.327% |
| 3 | ahv0v | 8293 | 3.99% |
| 4 | ap3m2;ahv0v | 7845 | 3.77% |
| 5 | apl4f;an3j0;an3j7;an3j8;apl4i;aojam;apkh5;apkh8 | 4364 | 2.1% |

# Inter Stream Interaction

When a bunch of client requests are concurrently processed by the server, there are possibilities of interaction amongst the client requests, {shared memory, cpu, io, file handle.} It is also imperative to know how good the Service is in terms of managing multiple threads of execution {mutex variables, semaphores, synchronization and coherence}.

These issues might be detrimental at times to either or both of the interacting client requests. We can approximate such an interaction between the client requests. We can hypothesize such as detrimental effect to their causation as opposed to correlation. In other words we might analyze possibilities 0f “it is because of this interaction between stream A and B there is an error now in stream B”. We approximate such an interaction by choosing two requests that happens in the “same time”. We claim that they are interacting if the follow the following predicate for requests A and B.

The following terminologies are analogy to a hypothesis where one EventStream will be affecting other EventStreams. An EventStream is ***Troubled*** if it is having the level field in one of the values {Warning, Unexpected, Exception, Monitorable, Critical}. A ***victimStream*** is an EventStream at focus. ***Neighbor stream*** are the ones EventStream which occurs at the same time as the victimstream as defined by the above predicate. We count the number of times the VictimStream was “troubled” “by” the neighboring streams and the number of the victim stream was not troubled by the neighboring streams when the victimStream is “interacting” with the Neighboring stream. We define “***Innocence***” of a Neighboring stream with respect to a victim stream as the ratio of the number of times the victim stream was not “affected” by neighboring stream to the number of times the victim stream. We declare a ***culprit stream*** with respect to a victimstream as the stream with least value of innocence. It all could be just be noise and the above definitions doesn’t mean anything as the probability of interaction among the request is quite less. We can verify them though just in case if we find anything interesting.



The following are the culprit streams for Roaming service.

1. *VictimStream*: 77a3;aebci;af7xc;aiwcp;akhz0;adhof;adhoi;af7xd;77a3

*CulpritStream*: 77a3;aebci;af7xc;aiwcp;akhz0;adhof;adhoi;af7xd;77a3

*Affected count*: 12

*NotAffected count*: 0

1. *VictimStream*: 77a3;aebci;af7xc;aiwcp;akhz0;adhof;adhoi;af7xd;77a3

*CulpritStream*: 77a3;aebci;af7xc;aiwcp;adhof;adhoi;af7xd;77a3

*Affected*: 12

*NotAffected*: 0

1. *VictimStream*: 77a3;aeqvg;aeqvs;adhoi;ajkyf;aeqvu;aeqvv;ajmah;aeqvv;ajmah;aeqvy;77a3

*CulpritStream*: 77a3;77a3

*Affected*: 11

*NotAffected*: 0

1. *VictimStream*: 77a3;aeqvg;aeqvs;77a3;77a3;aeqvu;aeqvv;ajmah;aeqvv;ajmah;aeqvx;a494;adhof;adhoi;adhog;adhof;adhoi;adhog;adhof;adhoi;adhog;adhof;adhoi;adhog;adhof;adhoi;adhog;adhof;adhoi;adhog;adhof;adhoi;adhog;adhof;adhoi;adhog;adhof;adhoi;adhog;adhof;adhoi;adhog;adhof;adhoi;adhog;adhof;adhoi;adhog;a495;77a3

*CulpritStream*: 77a3;77a3

*Affected*: 11

*NotAffected:* 0

1. *VictimStream*: 77a3;aeqvg;aeqvs;adhoi;ajkyf;aeqvu;aeqvv;ajmah;aeqvv;ajmah;aeqvy;77a3

*CulpritStream*: 77a3;ajmfo;77a3

*Affected*: 5

*NotAffected*: 0

When we investigate into stream 77a3;77a3 we found that “77a3 is the event that ULS logs when you start or stop a correlation”. It turns out to be noise.

Then there is not much information on the Not affecting Neighboring streams other than the fact that they represent the most frequently interacting streams. Below are the samples from Roaming services.

|  |  |  |  |
| --- | --- | --- | --- |
| Victimstream | NeighborStream | Affected | Notaffected |
| 77a3;77a3 | 77a3;ajmfo;77a3 | 1 | 13402 |
| 77a3;ajmfo;77a3 | 77a3;77a3 | 0 | 12811 |
| 77a3;ajmfo;77a3 | 77a3;ajmfo;77a3 | 0 | 4931 |

# Inter Project Dependencies

The notion is to give a better picture to the cryptic string of eventIDs in a human readable form. For an Onlooker, the event Stream doesn’t indicate anything clear, useful or significant.

**TagCode:**

Code written has logging lines which can be identified using a trace tag. Or an EventID which is an alphanumeric code. Each of these eventID uniquely identify the code. These tags can help identify source code line which may aid in bucketing failures. TagCode Database has the mapping between the trace tags and File Name/Line Number and date of tagging [18].

**Path Stream:**

If for every eventID in the EventStream, we substitute the File Directory (Top 2 keywords of the File path usually gives the area and the name of the Feature. It can be customized based on the needs if required) we get what is termed a ***Path Stream.***

**Component Stream:**

Since most of the project keywords are going to be repeated in a Tag stream, we would compact the TagStream and replace long repeating Project keywords with their frequency into what is termed as ***Component Stream.*** Well, basically we are doing a Run Length Encoding [19] on the event streams. A Component stream pretty much sums the interaction between the Projects while executing the service code. We can analyze what sort of “dependencies” are exhibited by the code. We can also find what sort of interaction among the projects leads to errors. Eg: An interaction between the services A,B with Service A->B->A->A->B->B could be having more errors than a version A->B->A->B->A.

We can gauge the reliability of an individual Project/Service/feature/Product in the part of the chain. We can also find the person in charge (:P)

Following are sample values of Tag stream and Bag Stream for Reader EventStreams.

***EventStream***: 77a3;aeqvg;aeqvs;al5ua;al5ud;al5ud;al5uc;aeqvu;aeqvv;ajmah;aeqvv;ajmah;aeqvx;a494;adhog;adhog;a495;

77a3

***Path Stream***:, util\ulsapi; msoserviceplatform\src; msoserviceplatform\src; readersvcs\exe; readersvcs\exe; readersvcs\exe; readersvcs\exe; msoserviceplatform\src; msoserviceplatform\src; msoserviceplatform\src; msoserviceplatform\src; msoserviceplatform\src; msoserviceplatform\src; msoserviceplatform\src; msoserviceplatform\src; msoserviceplatform\src; msoserviceplatform\src; util\ulsapi;

***Component Stream***: util\ulsapi (1); msoserviceplatform\src (2); readersvcs\exe (4); msoserviceplatform\src (10); util\ulsapi;

Instead of the cryptic eventID form, now the BagStream gives a clean view of the pathway of Projects involved in a client request. If we combine the before diagnostics and see through the glasses of the BagStreams we will have pretty cool stories to talk about.

# Event Stream Clusters and Advanced Diagnostics

There are questions such as

1. “What difference would it make if an eventID x is added to/modified/deleted from an EvenStream in terms of the health of stream?”
2. “Does similarity in the structure of the event stream means similarity in their error characteristics?”
3. ”Does structure alone determines the error?”
4. Which part of EventStream is responsible for the error in the stream?

Each of this questions need intense knowledge of the system in hand to be answered satisfactorily. Finding precise answers would be future work. However, we have taken the first step towards that direction.

We saw from the section of General diagnostics section from the sum of errors that the streams are expressing a really good similarity in the error characteristics when they have a structural similarity. We will make a good attempt at mining the structural similarity between the EventStream strings and verify them to their error characteristics.

**Finding similar streams, Clustering Streams**: The first step is to analyze structural similarity between the EventStreams. We can then query their error characteristics and analyze the results. We cluster the Event Streams based on string metrics into structurally similar groups. Currently the following are the string metrics chosen to measure how similar two event streams are to each other. For an event streams, we consider each of event ID as a single character in EventStream “String” and obtain the below metrics for a pair of EventStream strings.

1. ***BLAST score using Smith Waterman Algorithm:*** The Smith–Waterman algorithm performs local sequence alignment finding similar regions between two strings; Instead of looking at the total sequence, the Smith–Waterman algorithm compares segments of all possible lengths and optimizes the similarity measure [8].
2. ***String Edit-Distance or Levenshtein Distance***:

The Levenshtein distance between two words is the minimum number of single-character changes (insertion, deletion, substitution) required to convert one string into the other[9].

1. ***Longest Common Subsequence length:*** a **subsequence** is a string that can be derived from another string by deleting some characters without changing the order of the remaining characters. We find the LCSR length between two EventStreams [10].
2. ***Sorenson-Dice Coefficient***(bigrams) : This metric matches how good the bigrams of the strings matches. Bigrams are parts of the string of length two characters. We count the number of exact matches between the bigrams of both the string and divide by the half of sum of the lengths of the both the strings[11].

We perform clustering after finding the above distances between every possible event streams for Roaming and nexus services. Nexus has 94 unique Event Streams. Roaming has 353 unique streams. Reading has 2406. We are doing N\*(N-1) combinations of EventStreams to find the string metrics between all possible pairs of EventStreams.

Since the Length of the strings heavily impacts the score each the string pair get, we slightly modify the metrics to be length agnostic.

1. Blast Metric
2. LCSR Metric:

The Sorenson Dice coefficient is independent of length of the Strings. We choose only the strings of length Edit Distance 1. So we don’t need to calculate the Edit Distance Metric .

For a pair of EventStream strings A and B, we get the following scores.

We have construct the following table with rows having columns

1. EventStream A
2. EventStream B
3. Edit Distance
4. Lcsr Length
5. Sorenson-Dice Coefficient
6. BLAST score.
7. EventStreamA Length: The number of EventIds present in EventStreamA.
8. EventStreamB Length: The number of EventIds present in EventStreamB.
9. Blast Metric.
10. LCSR Metric.

***Clustering Algorithm***:

The rationale behind the algorithm is find the closest string pairss of least edit distance. Among them find the ones which has the highest local alignment scores given by blast metric. In each of those groups, find the string pairs having the highest bigram matches given by Sorenson-dice coefficient. Among those find the ones with longest common subsequence length value given by lcsr metric.Following are the thresholds and sorting criteria chosen of clustering the event streams:

1. ***Edit Distance: 1***

For now, We have taken the EventStreams which has the edit distance of 1 between each other.

This implies that with only one eventID change (insertion,modification,deletion),

We can convert one EventStream string into another.

1. ***Sort by Blast Metric (Descending order)***:

We choose the Event Stream string pairs of edit distance value 1 and sort them by the measure of Blast Metric in descending order. This measure ensures that we choose the ones with highest local alignment value.

1. ***Sort by Sorenson-Dice Coefficient value(Descending order)***:

We then sort them by Sorenson-dice coefficient to find the highest bigram matches.

1. ***Sort by Length of Longest common subsequence value(Descending order):*** Then we chose to find the longest common subsequence in their strings. The rationale is that the lcsr measure needs to have order of the characters to be maintained while the bigrams are less dependent on the order of the characters in a string. The blast metric finds the local alignment score which is additive compared to lcsr which is max of lengths, thus making it lesser important to use for string similarity.

The following is a sample result of clustering the Nexus streams:

***Area***: Nexus

1. EventStreamA: apl4f;an3j0;an3j7;an3j8;apl4i;aojam;apkh5;apkh8;ahv0v

EventStreamB: apl4f;an3j0;an3j7;an3j8;apl4i;aojam;apkh5;apkh8

Edit Distance: 1

Lcsr Length : 8

Sorenson-Dice Coefficient: 1.07692

BLAST Score: 15

StreamA Length: 9

StreamB Length : 8

Blast Metric: 0.0588

LCSR Metric: 0.4705

Corresponding behavior exhibited by the similar streams:

W: sum of Warning C: sum of Critical E: sum of Exception U: Sum of Unexpected M: Sum of Monitorable.

V: sum of Verbose, Me: sum of Medium, H: sum of High.

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Name | EventStream | V | Me | H | W | C | E | M | U |
| Stream A | apl4f;an3j0;an3j7;an3j8;apl4i;aojam;apkh5;apkh8;ahv0v | 0 | 24 | 0 | 0 | 0 | 0 | 0 | 0 |
| Stream B | apl4f;an3j0;an3j7;an3j8;apl4i;aojam;apkh5;apkh8 | 0 | 34912 | 1 | 0 | 0 | 0 | 0 | 0 |

Interesting observation here is that with an addition of one eventID (ahv0v), it changes the pathway and the number of medium levels is far lesser. While the regular path looks heavily used one with an level high as well indicating a serious event.

***Area***: Roaming

1. EventStream A: 77a3;aebci;af7xc;aiwcp;adhoe;adhoi;af7xd;77a3

EventStream B: 77a3;aebci;af7xc;aiwcp;**adhof**;adhoi;af7xd;77a3

Edit Distance: 1

Lcsr Length: 7

BLAST score: 13

Sorenson-Dice Coefficient: 0.83333

Stream A Length: 8

Stream B Length : 8

BlastMetric: 0.8125

Lcsr Metric: 0.4375

Corresponding behavior exhibited by similar streams.

W: sum of Warning C: sum of Critical E: sum of Exception U: Sum of Unexpected M: Sum of Monitorable.

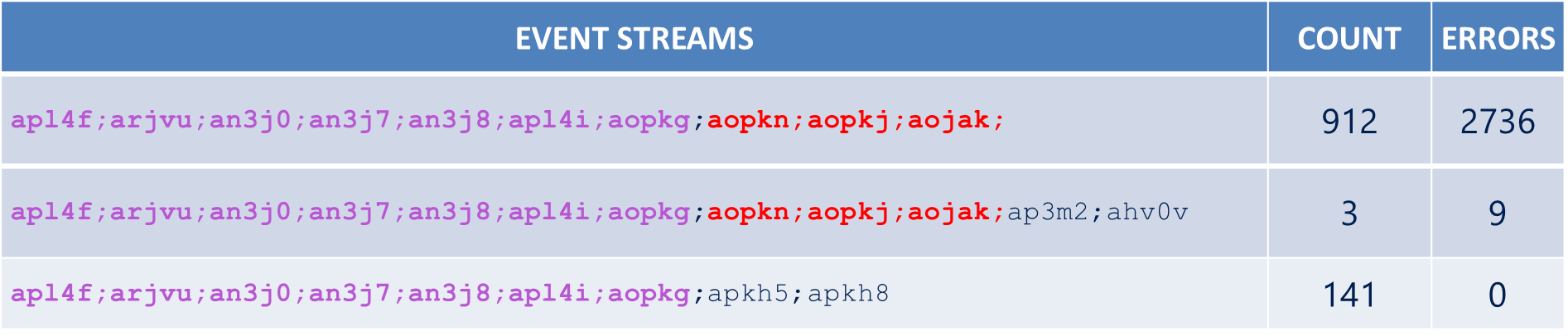
V: sum of Verbose, Me: sum of Medium, H: sum of High.

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Name | EventStream | V | Me | H | W | C | E | M | U |
| Stream A | 77a3;aebci;af7xc;aiwcp;adhoe;adhoi;af7xd;77a3 | 1131 | 1919 | 1 | 0 | 0 | 0 | 0 | 0 |
| Stream B | 77a3;aebci;af7xc;aiwcp;**adhof**;adhoi;af7xd;77a3 | 15 | 25 | 0 | 0 | 0 | 0 | 0 | 0 |

Here we see a similar observation where a change of eventID from “adhoe” to “adhof” makes a big impact the pathway usage. Using the string similarity clusters we were able to find the relationship between the streams and their pathway behaviors. We were able to answer the question1 in the beginning of the section.

To find relationships between the structural similarity of EventStreams and the similarity in the errors is just another query similar to above. (punted). We can find the closest streams to the current stream and spectate the data.

Question3 and Question4 needs more data and behavior analytics and developer’s attention to be answered satisfactorily.



**NEXUS EVENT STREAMS PATTERNS**

Even though there is a significant portion of the even streams are common to all the three event streams, their error behaviors are not the same. The first event stream follows up the pattern with an error and it is executed 912 times. While the second is same as the first one but has two events and it is executed only thrice. The third shows that the errors are independent of the pattern which was common to earlier event streams. This is a specific case for nexus. We could have dependency relationships between the event IDs on other services upon investigation.

# Limitations

There are bunch of caveats to the Evenstreams.

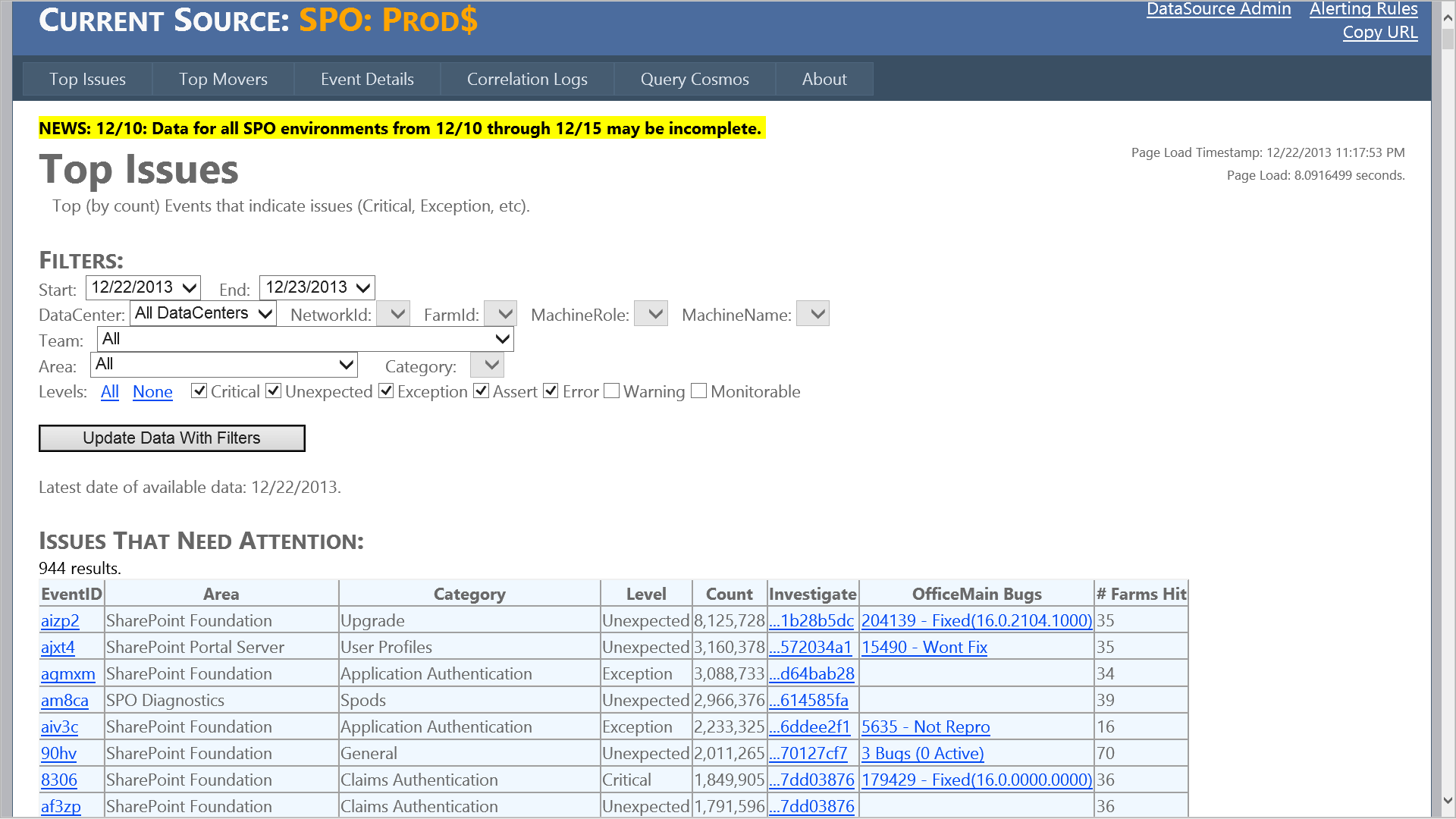
1. ***Length of the Stream***: Length of the Request is not easy to fix and certainly not predetermined. We would not know when the request “Ends”. It is difficult to classify a request into a stream without knowing whether the request has actually reached the end. While certain services do provide the notion of “start Correlation” and “End Correlation” in their message field it is not universal. Thus the classification is tentative for the time being. When a new event with the same correlation ID pops up, the corresponding stream diagnostics changes with it. Thus the system is not a static but a dynamic and stateless. we always specify time. It is consistent with the past though.
2. ***Correlation ID***: couple of things. Not every log line has correlation ID (a bug). Then, at times the correlation ID is not carried over throughout the entire session starting from the client’s request. But rather it is maintained like a tree which sprouts a new correlationID during the session whenever two servers interact with each other. New Id has no structural relationships with the earlier branch as essentially they are all GUID.
3. ***We(Office JETS team) don’t have ULS logs which connects across the Services.*** Currently the logs I had used are from same location and are not part of entire pathway of interaction between the client, the end Server and the intermediate servers.
4. ***Logging Order and Time Difference:*** The logging of the event need not be in the same order as the sequence of events happening. So there would be more number of Event streams than the actual number of possible observed pathways.

# Future Work

1. I have used only two days’ worth of ULS logs for 3 servers in a static manner. The idea can be extended to all Office Telemetry services and act as a live diagnostic service which maintains the EventStreams, their interaction, the BagStreams and produces charts on the fly.
2. ULS is a logging service. Likewise we have other signals such as Software Quality Metrics(SQM)[12] which pipelines the client usage data, Office Watson pipe which captures the crashes[13], Office Feedback which captures client’s feedback with a smile/frown[14]. We can combine the signals to have highly advanced diagnostics with which we can analyze the relationships between the crashes(Watson), errs in ULS, the sequence of commands used by the client(SQM) and user’s feedback in Smiles or Frowns(Office Feedback).
3. We can group the client requests in periodic manner to get a time series out of them. We can predict the total number of errs by time series analysis methods (Auto-Regressive Integrated Moving Average Methods[ARIMA])[15]. We can study the trends in the errors by using time series decomposition methods (Berlin V procedure[16]).
4. We can predict the eventstreams which would err by constructing a Levenshtein Automata which would choose the EventStream of known Levenshtein distance closest to the known set of erring EventStreams. This is assuming that the structural similarity heavily decides the erroring nature of the EventStreams. We can do the same for other string metrics we have collected [17].

# SUMMARY

Below is a screenshot of <http://officelogs/> where the Developers go and find the most important eventID errors, their frequency, the levels etl. This is obtained from the ULS logs to aid the developers to identify the most important issues.



The present system lacks the pathway diagnostics of execution of the services. It only presents the eventID which is most frequently hit by the services. This system does not have trends or data analytics. This Paper presents a powerful diagnostic tool which goes beyond debugging the current issues and provides a clear pathways of executions of the services.

* By capturing the pathways of execution (EventStreams), we are able to isolate the reasons that causes a trend of errors rather than a single error.
* All the possible observed pathways of execution per service are captured through EventStreams and are diagnosed for the most important, most delaying and most erroring pathways.
* The possible “interaction” between the pathways of concurrent requests are diagnosed to find the inter pathway issues. There are bugs which are related such issues.
* The similarity between the pathways shows similarity in the behavior, error characteristics and the usage of the pathways by the service. Clustering the EventStreams gives a good overview of the relationship between the structural similarity and the behavioral dynamics.
* The cryptic EventStreams were converted to a human readable form of Path streams and Component Streams.

# References

[1] [Exchange Server and Outlook](http://infopedia/kc02/docstore/_layouts/PowerPoint.aspx?PowerPointView=ReadingView&PresentationId=/kc02/docstore/Repository/Course%20A%20-%20Deploying%20and%20Supporting%20Office%20365%20Cloud%20Deployments/12%20-%20Exchange%20Online%20-%20Introduction.pptx&DefaultItemOpen=1)

[2] [Nexus Service](https://msft.spoppe.com/teams/jets/partners/SitePages/Nexus.aspx)

[3] [Cosmos](https://microsoft.sharepoint.com/teams/Cosmos/_layouts/15/start.aspx#/SitePages/Home.aspx)

[4]  [Roaming](http://officenet/11/teams/Roam/_layouts/PowerPoint.aspx?PowerPointView=ReadingView&PresentationId=/11/teams/Roam/Officenet%20Presentations/Office.net%20Roaming.ppt&DefaultItemOpen=1)

[5] [Reading](http://officelabs/projects/Reading/_layouts/PowerPoint.aspx?PowerPointView=ReadingView&PresentationId=/projects/Reading/Shared%20Documents/Technical%20Teardown/Office%20eReader%20(Michael%20Cooper%27s%20Team).pptx&Source=http%3A%2F%2Fofficelabs%2Fprojects%2FReading%2FShared%2520Documents%2FForms%2FAllItems%2Easpx%3FRootFolder%3D%252Fprojects%252FReading%252FShared%2520Documents%252FTechnical%2520Teardown%26FolderCTID%3D0x012000CA99F64465D93245AC0473FFD8685799&DefaultItemOpen=1)

[6] [Unified Logging Service (ULS)](https://microsoft.sharepoint.com/teams/uls/documentation/Home.aspx)

[7] [Correlation ID](https://microsoft.sharepoint.com/teams/uls/documentation/Home.aspx)

[8] [Smith-Walterman Algorithm](http://en.wikipedia.org/wiki/Smith-Waterman_algorithm)

[9] [Levenshtein distance](http://en.wikipedia.org/wiki/Levenshtein_distance)

[10] [Longest common subsequence problem](http://www.ics.uci.edu/~eppstein/161/960229.html)

[11] [Sorenson-Dice Coefficient](http://en.wikipedia.org/wiki/S%C3%B8rensen%E2%80%93Dice_coefficient)

[12] [Software Quality Metrics](http://sqmteam/default.aspx)

[13] [Watson](http://watson/)

[14] [Office Feedback](http://ofeedback/)

[15] [Auto-Regressive Integrated Moving Average method for time series prediction](http://people.duke.edu/~rnau/411arim.htm).

[16] [Berlin V Procedure](http://www.google.com/url?sa=t&rct=j&q=&esrc=s&frm=1&source=web&cd=8&cad=rja&ved=0CGoQFjAH&url=http%3A%2F%2Fwww.researchgate.net%2Fpublication%2F5160071_A_robust_data-driven_version_of_the_Berlin_Method%2Ffile%2F60b7d5170001f926f6.pdf&ei=qnS2Uty5LtjeoAT2h4DIBw&usg=AFQjCNHF6AHzGBbSNZBicoLsPDOHqOAYkA&sig2=MvoJXISjNvBAeHv22cZNqw)

[17] [Levenshtein Automata](http://en.wikipedia.org/wiki/Levenshtein_automaton).

[18] [TagCode](https://msft.spoppe.com/sites/howto/reliability/Wiki/Assert%20and%20ULS%20Tagging.aspx)

[19] [Run Length Encoding](http://www.princeton.edu/~achaney/tmve/wiki100k/docs/Run-length_encoding.html)

Go ahead and get started. !