March 2023

A MECHANISM TO CORRELATE DISTRIBUTED LOGS TO PERFORM ROOT CAUSE ANALYSIS AND SCOPE ASSESSMENT

Xueqiang (Sherman) Ma

A MECHANISM TO CORRELATE DISTRIBUTED LOGS TO PERFORM ROOT CAUSE ANALYSIS AND SCOPE ASSESSMENT

AUTHOR:
Xueqiang (Sherman) Ma

ABSTRACT

For large distributed systems, it can be difficult to connect isolated types of data across multiple data logs, as well as to determine correlations among the data logs in order to perform root cause analysis during service outages such that a scope of damage can be assessed. Presented herein is a mechanism to correlate distributed data logs in order to perform root cause analysis and scope assessment. The mechanism may be utilized to analyze and resolve issues in large, complex distributed and/or cloud systems that generate high volumes of data logs.

DETAILED DESCRIPTION

Large distributed systems typically involve tools and processes to collect logs, traces, and metrics from multiple components of the systems. However, connecting these otherwise isolated types of data to effectively navigate between logs is a challenge, as is finding correlations to perform root cause analysis during service outages and assess the scope of damage of such outages. To efficiently handle the vast and ever-growing data, it is important for analysis algorithms to be highly efficient in order to ensure prompt response times.

Consider, Figure 1, below, which illustrates a reference model for distributed log analytics based on a Kubernetes® (K8s®) cluster. Kubernetes® and K8s® are registered trademarks of the Linux Foundation.

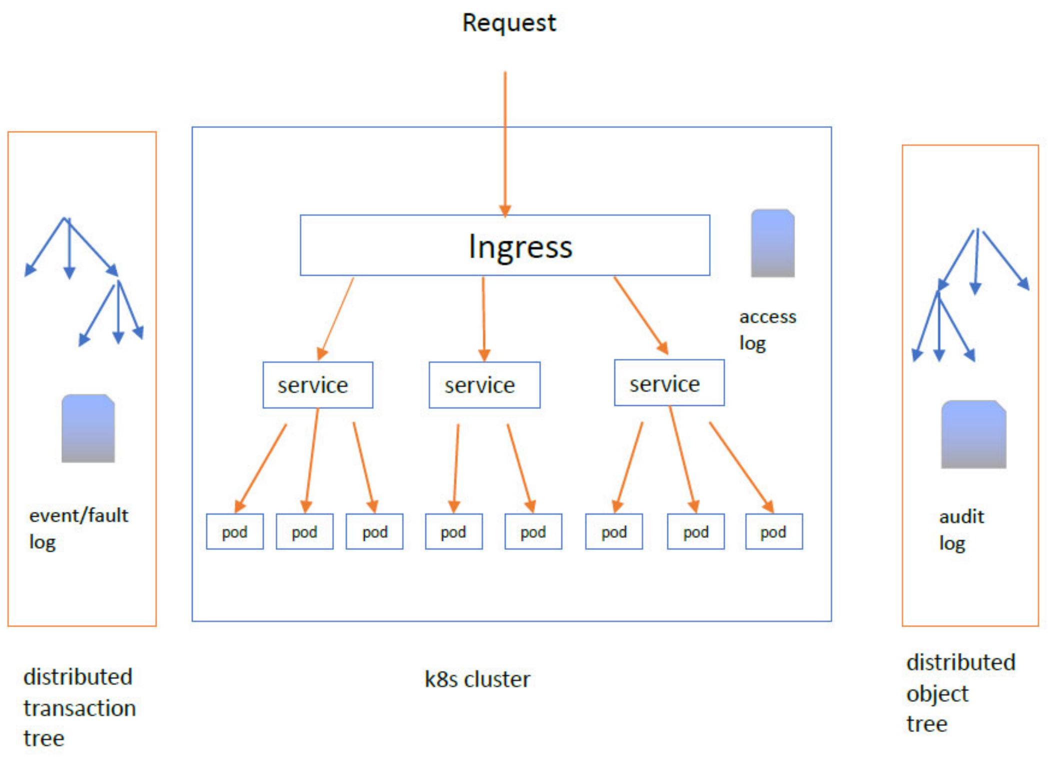

Figure 1: Example Reference Model based on a K8s Cluster

As illustrated in Figure 1, Representational State Transfer (REST) requests can be received at an Ingress, where an access log can be created for each configuration or query request. Each request can then be dispatched to different services. These services can, in turn, invoke various container-based microservices to complete a request handling transaction and update data in a persistence object store for stateful transactions.

The persistent object store is often structured as a logical distributed object tree (DOT), such as a datastore, resource tree, or any registry built on an open-source distributed database. When each object is updated, an audit log entry is created to record the changes, including time stamp and other relevant information. Figure 2, below, illustrates an example DOT.

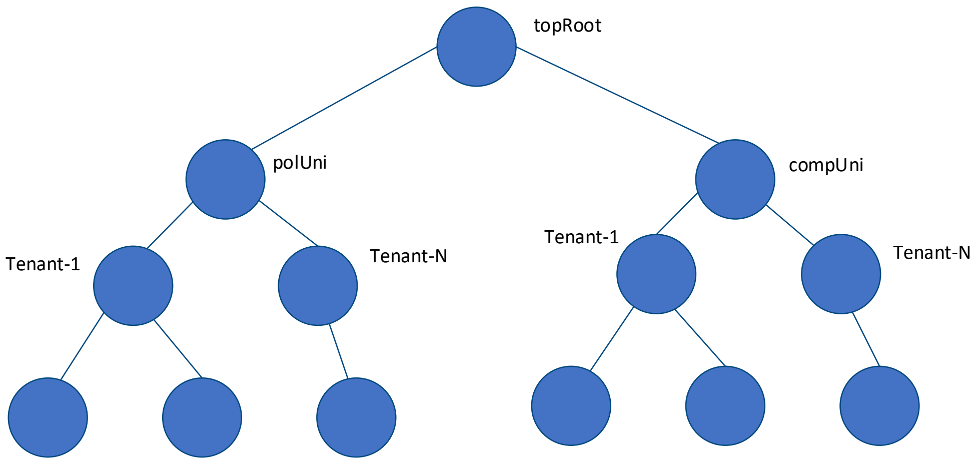

Figure 2: Example DOT

During operation of the example Figure 1 reference model, every REST request initiates a request transaction at the Ingress point, which triggers additional transactions by accessing Application Programming Interface (API) entry points of other service(s) through REST APIs. These APIs, in turn, calls APIs on other services or Point of Delivery (POD) network modules. In a similar manner to how data is abstracted, the distributed transactions can be represented as a Distributed Transaction Tree (DTT). An example, DDT is illustrated below in Figure 3.

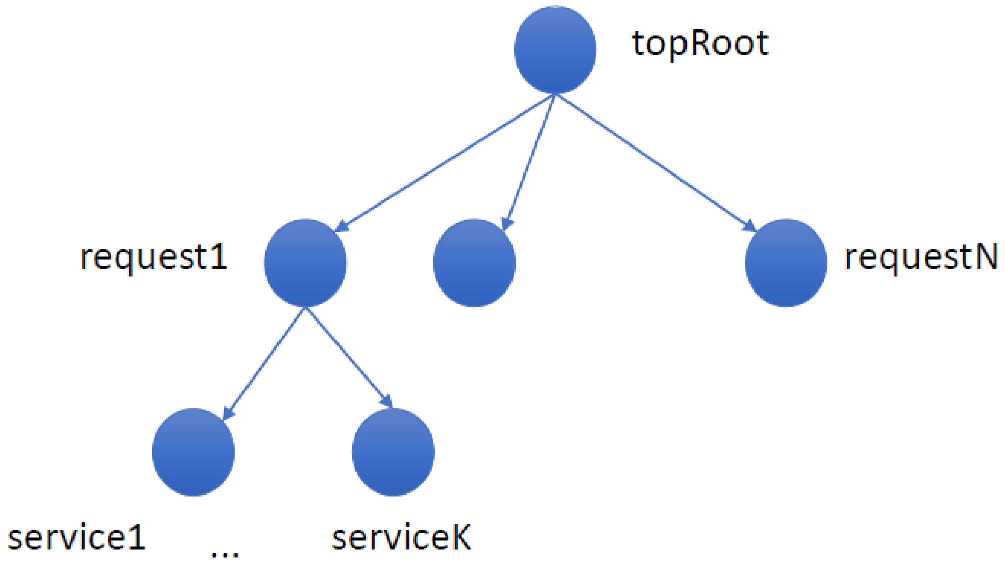

Figure 3: Example DDT

Figure 3, above, illustrates a logical DTT in which the top root represents the Ingress process itself. The second-level children represent the request handler transactions that the Ingress process launches. Each request handler invokes its service calls as sub-transactions and so forth. These transactions can be running on different PODs or worker nodes, creating a logical distributed tree.

Each transaction in the DTT has a unique transaction ID (TID). The DTT captures the relationship between TIDs, such as which TIDs are the children of a given TID. Given a TID, the tree can be traversed to determine all its ancestor transaction nodes and all its descendent transaction nodes. In addition to the TID, a timestamp can also be stored that indicate when the transaction was launched.

To track the transaction relationship in a distributed system such as a K8s cluster, a dedicated Hypertext Transfer Protocol (HTTP) header can be provided, identified as: 'http-parent-tid'. This header helps to propagate the TID and the time when a current transaction invokes remote APIs, which provides for the ability to build the DTT in a distributed manner.

New metadata can be added in the following three types of logs to facilitate intelligent log correlation and analytics in order to perform root cause isolation and damage scope assessment, as follows:

- Access Log: When a REST API request is received from the Ingress, an access log entry is created that records the request header and body, the request's client source Internet Protocol (IP) address, the requester's username and access role, the time stamp of the request, etc.

- Audit Log: When an object in a DOT is updated within a transaction, an audit log entry is created that records the updated object in the DOT, its creation/modification/deletion status, and the time of the update.

- Event/Fault Log: Faults and events can be defined as conditions represented by a Boolean expression based on the properties of objects. When an object is updated, the defined conditions will be evaluated to check whether a condition value has been changed. If a condition value changes from false to true, a fault/event will be raised. If a condition value changes from true to false, the raised fault will be cleared, but a fault record entry will be created as a history record.

To help navigate between different types of logs, additional metadata can be provided that relates a TID to the three types of logs mentioned above so that each log entry can be linked to the transaction it created and the node context in DTT, as follows:

- Access Log: the TID corresponding to the request handler transaction at the second level of the DTT will be added.

- Audit Log: besides the object ID (sometimes referred to as a Distinguished Name (DN)), the TID of the transaction performing the object update will also be added.

- Event/Fault Log: besides the object ID, the TID of the transaction that made the object update to trigger the fault condition change will also be added.

During operation when a fault is raised, the path can be traversed upward using the TID in the fault log entry on each ancestor node along the path. The TID of the node and the timestamp can be used to look up the audit log and determine which transaction directly or indirectly caused the fault, as well as which objects were modified. When the 2nd level of the DTT is reached, the TID and timestamp can be used to look up the access log and determine the exact client (source IP, username, and access role) that initiated the request.

It is noted that logs may be stored in an append-only data structure in increasing order of timestamp such that, given a timestamp, a binary search can be used to locate entries with matching timestamps in O(*logN*) time.

When two or more similar faults occur somewhere in the system, there is a chance that they are triggered by the same cause. To find the root causes and fix them at their source, assume, for example, that there are *m* nodes (*node1, ..., node*$_m$) that share similar faults in the DTT, and these nodes don't have to be at the same level of the tree. If the depth of each node is precomputed as, depth($node_i$), and is stored in the DTT, a procedure as illustrated in Figure 4 can be used to locate the root cause node(s) for all these faults.

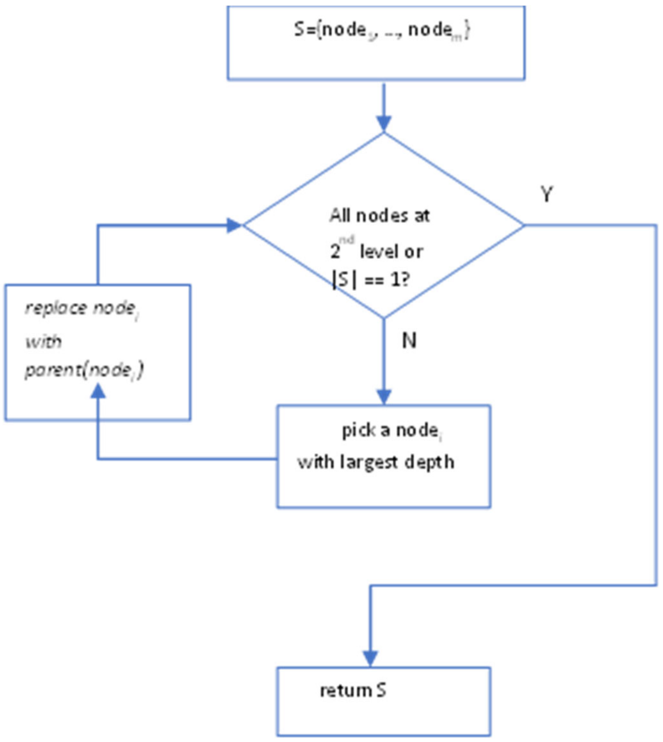
S={node$_1$, ..., node$_m$}
All nodes at 2nd level or |S| == 1?
Y
N
replace node$_i$ with parent(node$_i$)
pick a node$_i$ with largest depth
return S

Figure 4: Example Procedure to Calculate Root Cause Nodes

Since in each iteration, one of the lowest nodes from the set S can be picked and moved one level up and if its parent node is also in the set, it will be merged into its parent node. This process will eventually stop when all nodes have reached the second level or there is only one node in the set. On average, the procedure stops in O(*log(N)*) iterations.

If the process ends with only one node in the set S, this node is considered the root cause node of all the faults, otherwise the faults are caused by multiple requests, which typically indicates a more coordinated attack than an isolated misconfiguration.

Yet another interesting application is to assess the scope of a damage when a set of faults occurred. First, the number of nodes in DTT exhibiting those faults can be used as the indicator of the scope, but that does not capture the useful information as whether the faults are happening at neighboring nodes within same location, whether only nodes at the same level are affected or nodes at any level will be affected, and whether only nodes close to the root cause node are affected or nodes deep in the tree are affected as well. This information is very useful for postmortem analysis.

Therefore, to better understand the extent of the damage, a set of parameters can be established to characterize its scope, as follows:

- Node Distribution Number: All the given faults can be mapped to the set of nodes based on the TID in fault log entry such that the node distribution number can be defined to be the size of the resulting TID node set.

- Width: The procedure illustrated in Figure 4 can be executed to return a final set of nodes. If there is only one node in the final set, the damage can be determined to be caused by a single transaction. If the set contains multiple second level nodes, then it can be determined the system was attacked by a set of requests. The width parameter can be defined to be the size of the final set.

- Depth: The depth of the damage can be defined to be the largest depth of all the faulty nodes, which can indicate how deep the damage has been propagated.

- Span: The span can be used to measure the extent of the damage's reach in the tree. The span can be calculated as follows:
    - Find all the affected nodes with the largest depth, denoted as S1, pick the left most node A from S1, if there are more than one node in in S1, then pick the largest node B from S;
    - If there is only one node in S1, find all the affected nodes with the second largest depth, denoted as S2, pick the largest node B from S2;
    - Find the least common ancestor of A and B, denoted as RC. Note that RC can be the top root node;
    - Calculate the length of the path between node A and node B as the damage span, for example, path (A, B) = path (A, RC) + path (RC, B).

It can be shown that the above parameters can be calculated in O(*logN*) on average.

As discussed above, it can be assumed that the DTT has a precomputed depth and parent pointers stored, which can be costly, especially when the tree grows large. Upon closer inspection, it has been determined that all nodes except the top root node, which acts as a demon process continuously accepting requests and therefore can have many child nodes, have a limited number of children. This is because each request is to receive a response within a certain time period, or it will time out. As a result, for each subtree with its root at the second level, it can be assumed it has no more than k child nodes, making it a *k-ary* tree (it may contain fewer children than k). Figure 5, below, illustrates a general request *k-ary* subtree of k and a transaction auxiliary (TXID) assignment.

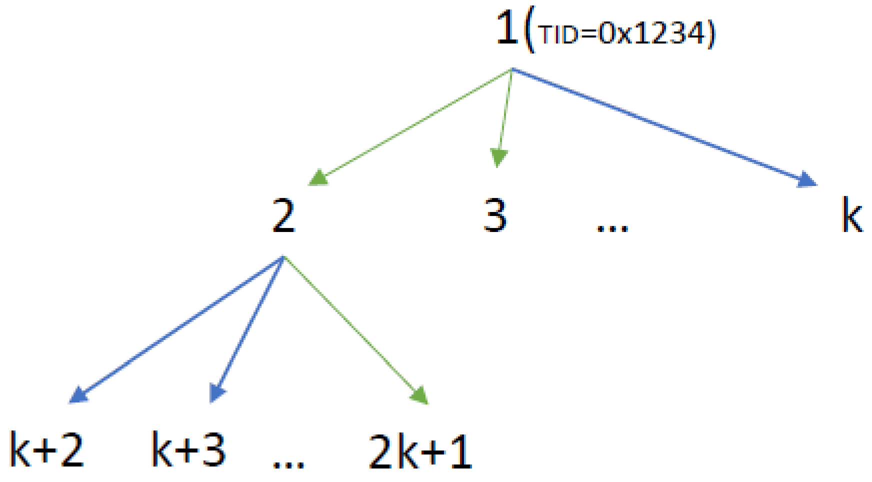

Figure 5: General Request k-ary Subtree of 'k' and TXID Assignment

As illustrated in Figure 5, a subtree with the query node TOD=0x1234 is shown, along with the assigned TXIDs at each node. For each of these *k-ary* trees, a TXID can be assigned, as follows:

- The root of a *k-ary* tree will have a TXID of 1.
- If a parent node's TXID is p, its children's TXIDs will range from (p-1)k+2 to (p-1)k+2k- 1.

It has been shown that for a child node with TXID c, its parent's TXID is p = [(c-2)/K] + 1 and its depth is [$\log_k$ c] + 1.

With the introduction of TXID, each node at the second level or below can be uniquely identified by its 2nd level request's TID and its local TXID. TXID can be represented as a 64-bit number, therefore, even with a large value of k, such as 64K (2^16), it can still represent a DTT tree with a maximum depth of 48 (64-16).

When calculating the root cause node of a set of faulty nodes within a request, a common use case, it is no longer necessary to access the global DTT to determine a node's depth and parent. The calculation of a parent's TXID can be reduced to a simple numerical calculation. Since the depth of the original DTT is at most 48+1, the algorithm can be performed in constant time.

Given the TXIDs of node A and node B in the definition of span, the calculation of the damage scope span parameter between them is illustrated in Figure 6, below.

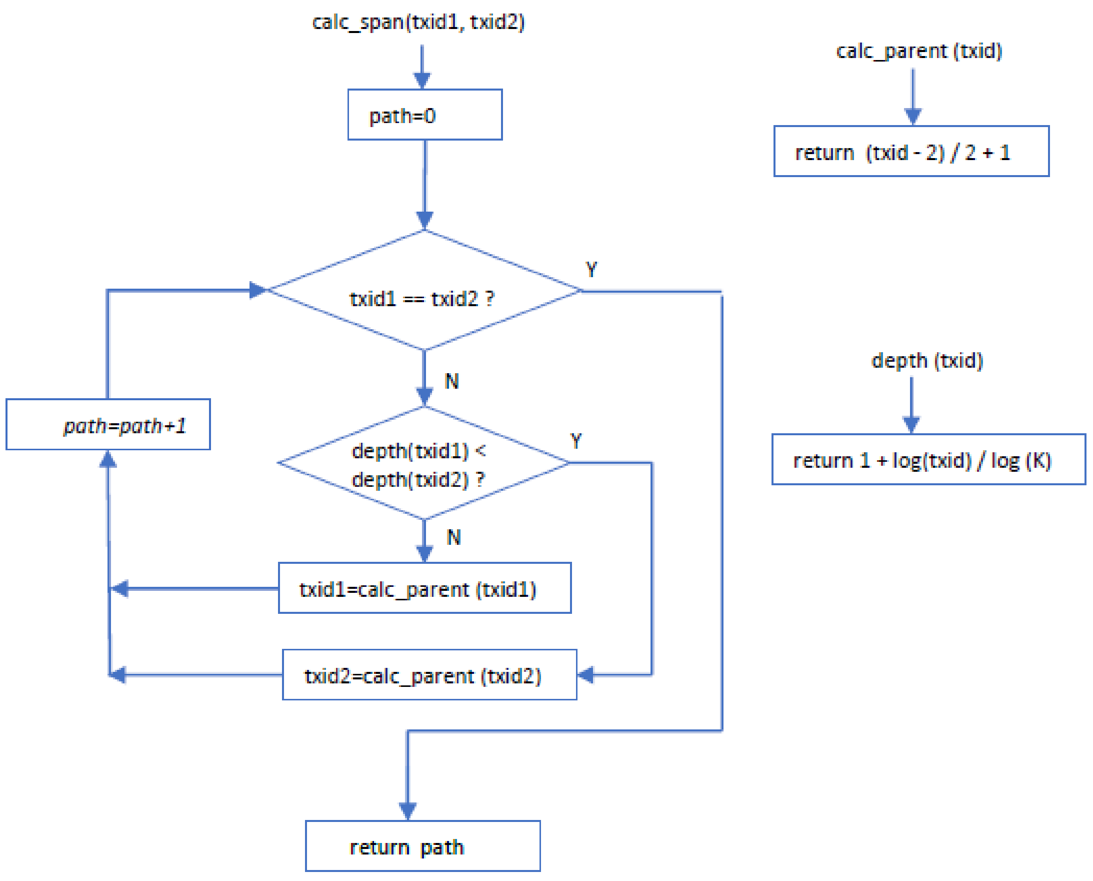
calc_span(txid1, txid2)
path=0
txid1 == txid2 ?
Y
N
path=path+1
depth(txid1) < depth(txid2) ?
Y
N
txid1=calc_parent (txid1)
txid2=calc_parent (txid2)
return path
calc_parent (txid)
return (txid - 2) / 2 + 1
depth (txid)
return 1 + log(txid) / log (K)

Figure 6: Example Procedure to Calculate the Span Between Two TXIDs

In summary, this proposal provides a mechanism through which a TID can be propagated using a dedicated HTTP header in order to construct a distributed transaction tree (DTT). Further, access, audit, and event/fault logs can be enhanced by incorporating

TID metadata, which can enable tracing and linking different log types using a reference model along with an algorithm-based solution. Additionally, an efficient O(*logN*) procedure can be adopted to identify common root causes and damage scope (such as node distribution, width, depth, and span) for sets of faulty nodes with similar faults in the DTT. To improve performance within a request sub-tree, a 64-bit TXID can be introduced for the k-ary tree that will allow for the calculation of root cause and damage scope parameters with TXIDs instead of traversing a DTT.

Accordingly, the solution provided herein is ideal for analyzing and resolving issues in large, complex distributed and cloud/k8s systems that generate high volumes of logs. Moreover, practical solutions are provided that can be realized through real-world root cause analysis and damage scope assessment problems.