BlackBoxesDebug

Monday, April 11, 2016 10:50 PM

分布式系统通常拥有很多的通信组件，因此其调试是非常困难的。尤其是当系统体现出了一种贫乏的性能时，而且通常这种系统都由来自不同的厂商的组件所构成，而且没有源代码。传统的方案提供商的员工并不一定对这种系统的调试很有经验。因此需要设计一个工具来辅助技能不突出的程序员独立地找到性能的瓶颈。为此，Marcos K. Aguilera在《Performance Debugging for Distributed Systems of Black Boxes》一文中，提出一个工具帮助程序员在这种的黑盒分布式系统中隔离性能瓶颈。工具通过对和应用无关的被动的跟踪不同结点间的消息流，并对这些跟踪数据offline的分析。

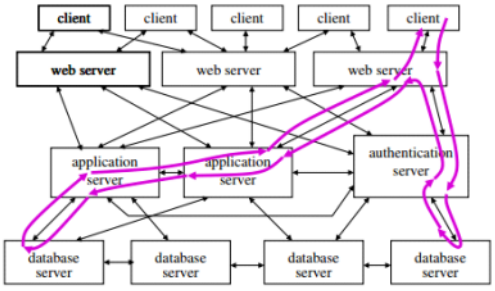

Figure 1: Example multi-tier application showing a causal path

图是一个典型的分布式的多web服务器系统。紫色的箭头标明一个可能的消息流。其中的网络延迟在此处认为可以忽略。这篇文章中提到的方法也同样适合基于消息的系统。最终的目的是能够帮助程序员了解并追踪性能延迟的来源。

找到影响力大的路径。

找到高延迟的节点。

为了能够在纷繁复杂相互掺杂的消息中找到对应的路径，同时得出统计信息，本文提出的方法分为三个部分：

1. 暴露并追踪通信
2. 推测关联路径
3. 可视化

第五章作者介绍了第二阶段的分析算法。

算法的主要挑战在于如何推测关联的路径，以及从关联的消息中推测延迟。为此作者实现了两个算法，一个为了RPC-style系统而实现的算法，另一个则不限定消息机制。

1. 嵌套算法
   该算法将所有的边两边的消息合并成一个文件，并且显式地验证调用之间的嵌套关系。自然，这种嵌套关系的推导需要对应的系统是基于RPC机制实现的。当然一对调用可能仅仅是在时间上是嵌套的，但是如果大量重复出现这种嵌套，那么可以推断两者在逻辑上也是嵌套的。而节点之间的延迟可以通过简单的从不同节点的消息的时间戳上得出。
2. 回绕算法
   和嵌套算法单独考察每条消息不一样的是，回绕算法考虑多个消息的聚合来寻找相关关系。算法将整个系统的trace按照边的两头分成一系列的traces，并且将每个消息当作一个时间信号看待。然后通过信号处理技术找到不同的信号之间的共同关系。从A到B之间和B到A是不同的信号。通过对信号的读取，构造相应的路径。

为了获得所有节点上的trace，根据之前的黑盒假设，每个组件的内部并不重要，而是需要每个消息的发送者，接收者，和时间戳。如果是嵌套算法，还需要知道这个消息是调用还是返回。为了获得不同层次的消息信息，并且不依赖于应用程序，作者开发了被动网络监听，中间件构件，kernel构件和应用程序构件。

1. 被动网络监听， 不需要修改现有的系统，对系统的运行没有影响，因此是首选的方案，具体的实现包括Port Mirroring和数据包嗅探。
2. 追踪J2EE系统，因为J2EE系统有自己的调用工具，因此可以基于工具的消息之上进行处理
3. 应用层，通常不需要这个层次，但是如果应用程序有提供，那么也可以使用。

实验和实验结果：

1. 基于自己的tracelet的实验
2. J2EE实验
3. 接收头跟踪

为什么适合做Kernel的开发

1. kernel调试很复杂，对程序员的水平要求很高
2. 黑盒测试可以避开大量的内核相关的代码

对我的启发：

1. 适合Kernel，不需要了解细节
2. 确实，如何定义一个继承式的调用结构中的节点之间的时间，是解决这个问题的关键。

```
1.  procedure FindPathsFromRoot(i)
2.  T_i := trace of messages with source i
3.  output_graph := graph with one vertex x_i labeled i
4.  for each destination node j in T_i do
5.      V := messages in T_i with destination j
6.      add vertex x_j labeled j and edge (x_i, x_j)
7.          to output_graph
8.      ProcessNode(j, x_j, V)
9.  procedure ProcessNode(j, x_j, V)
10.     T_j := trace of messages with source j
11.     O_1, ..., O_m := FindCausedMessages(V, T_j)
12.     for n := 1 to m do
13.         k := O_n.node;  W := O_n.messages
14.         d := O_n.delay
15.         add vertex x_k labeled k and edge (x_j, x_k)
                labeled (|W|, d) to output_graph
16.         ProcessNode(k, x_k, W)
17. function FindCausedMessages(V, Z)
18.     m := 0
19.     C := FindCorrelation(V, Z)
20.     find positions of spikes of C(t)
21.     for each spike position d found do
22.         Z' := messages in Z with a timestamp equal
                to some timestamp in V shifted by d ± v
23.         for each destination node k in Z' do
24.             m := m + 1
25.             O_m.node := k;  O_m.delay := d
26.             O_m.messages :=messages to k in Z'
27.     return O_1, O_2, ..., O_m
28. function FindCorrelation(V, Z)
29.     s_1 := indicator function for V
30.     s_2 := indicator function for Z
31.     return cross_correlation(s_2, s_1)
```