Application Dependency Tracing for Message Oriented Middleware

Li-Juin Wu, Hong-Wei Li, Yu-Jui Cheng, and Yu-Sung Wu*
Dept. of Computer Science
National Chiao Tung University
Hsinchu City, Taiwan
lee987667@hotmail.com, g6_7893000@hotmail.com, chengyj@cs.nctu.edu.tw, hankwu@g2.nctu.edu.tw

Houcheng Lin
Cloud Computing Center for Mobile Application
Industrial Technology Research Institute
Hsinchu County, Taiwan
linhaocheng@itri.org.tw

Abstract—**Software defined infrastructure greatly reduces the deployment cost of distributed applications. Many distributed applications employ message oriented middleware (MOM) for the integration of heterogeneous components and to achieve scalability and fault tolerance. The structure of a distributed application can be very complex. In addition, the asynchronous message delivery model of MOM further complicates the runtime behavior of a distributed application. To diagnose a faulty distributed application, one often needs to determine the dependences of its messages, and by extension, the dependences of its components. We propose Message Tracer to identify the message dependencies of a MOM-based distributed application. Message Tracer sniffs the network traffic of MOM and uses knowledge of message broker protocols to establish the dependencies. Message Tracer makes no assumption on the application threading model and incurs negligible performance overhead. Message Tracer correctly identified 95% of the dependencies for the common use cases and 75% of the dependencies when the system was under extreme stress.**

Keywords—application dependency, distributed application, message queue, message broker, tracing, and testing

I. Introduction

Software defined infrastructure (SDI), such as datacenters that employ platform virtualization and software defined networking technologies, significantly lowers the deployment cost of large-scale distributed applications. A large-scale distributed applications is typically made up of heterogeneous components interconnected by message oriented middleware. Due to the distributed nature, this kind of applications are inherently difficult to debug. When diagnosing performance or functional issues of a distributed application, one often needs to first determine the dependencies of its constituent components at the moment when the issues were perceived. This has aroused interests in the research of distributed application dependency tracing [1-4]. However, the use of message oriented middleware and third-party components in modern large-scale distributed applications limits the effectiveness of these existing solutions due to more flexible threading models and lack of source code access to the third-party components.

In this paper, we propose Message Tracer to determine the dependencies of the messages communicated through message oriented middleware by the components of a distributed application. Message Tracer uses packet sniffing to observe the network traffic of message broker services and rely on knowledge of the message queue protocols to identify the dependences of the messages. Message Tracer does not require source code of the application components. It incurs minimal performance overhead and can attain high tracing accuracy with the help of a novel technique to suppress the impact of packet loss during packet sniffing.

In the following, we will give a brief overview of distributed application dependency tracing in Sec III. The design of Message Tracer is presented in Sec. IV. An evaluation of the Message Tracer prototype is given in Sec. V followed by the conclusion in Sec. VI.

II. Related Work

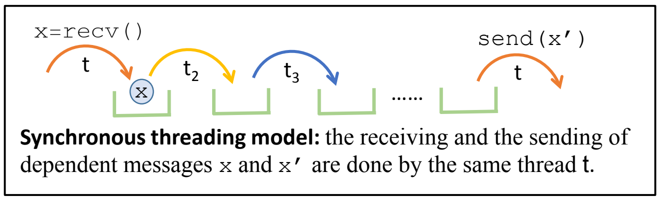
There are three existing approaches to provide application dependency tracing. The first approach is based on source code level instrumentation, which inserts monitoring code at key locations (i.e. code locations where messages are sent / received) in the application source code [2, 3, 5]. This approach can identify dependencies at the granularity of messages and incurs little overhead. However, due to the requirement of source code modification, the approach is not generally applicable to third-party software components.

The second approach is based on monitoring the network API invocations made by the application components. Through correlation of `send` and `recv` API invocations, solutions such as [4, 6-8] can identify the dependencies without modification of the application source code. The approach can identify message-level dependency. The performance overhead is higher than the first approach. The major limitation of the approach is that it requires the application components to follow the synchronous threading model, that is, the receiving and sending of causally related messages have be carried by the same thread, and the thread cannot be involved in the sending or receiving of other unrelated messages at the same time. This type of approaches is therefore not suitable for applications that employ MOM, as the operations of MOM do not follow the synchronous threading model.

The third approach is based on statistical inference of the application network traffic data [1]. It relies on the assumption that interactions between dependent application components will have distinguishable patterns in the network traffic. This approach incurs little overhead and is non-intrusive to the applications. However, they generally provide component level dependencies only in contrast to the message level dependencies as provided by the previous two approaches. The approach's accuracy can also be negatively affected if application does not synchronous threading model. Overall, it is still not a good solution in tracing application dependency for message oriented middleware (MOM).

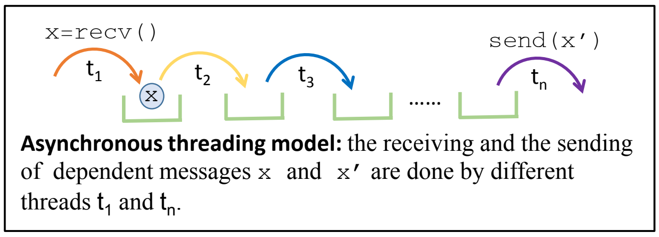
III. Distributed Applications in Software Defined Infrastructure

Platform virtualization and software defined network (SDN) technologies allow the construction of datacenters that provides a cost-effective infrastructure for running large-scale distributed applications. A large-scale distributed application is likely to be consisted of heterogeneous components. Each individual component inherently has its own unique characteristics. Both the workload level and processing speed can be different for each component at any moment. Occasionally, a fast running producer component may need to wait for a slow running consumer component to complete the current task before the producer component can proceed to the next task. This would result in low resource utilization at the producer component. The issue can be addressed by incorporating message oriented middleware (MOM) [9] as the interconnection of the application components. The middleware can buffer the output of the fast component. On the other hand, one may arrange parallel instances of the slow component to speed up its processing. The middleware can also serve as the aggregator (an abstraction) of the slow component instances.



Synchronous threading model: the receiving and the sending of dependent messages `x` and `x'` are done by the same thread t.

Figure 1. Synchronous threading model



Asynchronous threading model: the receiving and the sending of dependent messages `x` and `x'` are done by different threads t_1 and t_n.

Figure 2. Asynchronous threading model

A. *Application Dependency Tracing*

Distributed applications are inherently more complex than monolithic applications that run on standalone machines. The interactions among components of a distributed application can be obscure and ever-changing. When debugging such a distributed application, it often requires one to first identify the dependencies among the components and then continue to track down the causes of the issues. Component dependency is a relation between two components when the execution of one of the component depends on the output of the other. For example, when the component *caller* makes a remote procedure call (RPC) to the component *callee*, there exists a dependency *caller*→*callee* as the *callee*'s execution depends on the RPC call made by the *caller*. For a distributed applications, there are multiple dependencies among its components. By tracing the dependencies, one can have an overview of the states of the distributed application. From there, diagnosis of the application can be conducted so that the root causes of an error can be pinpointed.

The identification of application dependencies typically starts by establishing the dependencies of the messages received and sent by each component. The *per*-component dependencies can then be merged to give the overall application dependencies. We are mostly interested dependency tracing techniques that can provide dependency at the message level and do not require source code modification. This has been attempted by previous work[4] by assuming that the receiving and sending of dependent messages are carried by the same thread (Figure 1). Specifically, one can hook the `recv()` and `send()` APIs, and assume that `x'` is dependent on `x` if the receiving of `x'` "`x'=recv()`" is the nearest `recv()` invocation before the invocation of `send(x')` in the same thread. However, this technique would not work for applications that employ MOM ware, as MOM commonly adopts the asynchronous threading model (Figure 2), where the receiving and sending of messages may not be done by the same thread.

B. *Message Oriented Middleware*

MOM serves as generic medium to integrate heterogeneous components. MOM also provides asynchrony that can accommodate variations of workload or processing power across the components of an application. The most popular incarnation of MOM is a message broker, or commonly referred to as a message queue service, where application components can store and deliver messages in an asynchronous manner. Some example brokers that are widely in use in datacenter environments include IBM WebSphere MQ[10], Apache ActiveMQ[11], RabbitMQ[12], and etc. Different brokers typically implement different message protocols, but most of the protocols follow Java Messaging Service (JMS) standard. The Message Tracer prototype is built around Apache ActiveMQ, which implements both OpenWire and AMQP message protocols.

1) Jave Messaging Service

Java messaging service (JMS) [13] is a message queue abstraction standard defined by Oracle. JMS specifies the essential methods and message fields required for the communication with message queue. There are many message queue protocols that implement JMS standard including OASIS AMQP [14], Apache OpenWire [15], IBM WebSphere MQ JMS, and etc. In JMS, there are two transmission models. One is the point-to-point model. In this model, a message put by a sender node to a message queue can be retrieved by a single receiver node, and the messages between each pair of sender and receiver nodes are delivered in order. The other model is the publish-subscribe model. It is similar to newsletter. If a node publish some message to the message queue with a topic, and any nodes that subscribe to the topic will receive a copy of the message from the message queue.

2) AMQP

OASIS Advance message queue protocol (AMPQ) is a popular protocol for message queue communication. The current version is AMQP 1.0, which is completely different in design from past versions of AMQP. In AMQP 1.0, a message is referred to as a *frame* and is transmitted in binary form. A *frame* contains five elements, which are *frame size*, *data offset*, *frame type*, *channel*, and *payload*. The *payload* consists of one or more descriptors and can be structured as a hierarchy. Each descriptor has its own code. Therefore, a frame can describe several parameters, and a broker using AMQP can define its own descriptor type to support non-standard functionalities. This design makes AMQP very extendible and efficient.

3) OpenWire

OpenWire is the native message queue protocol supported by ActiveMQ. Under OpenWire, each message is referred to as a command, which consists of a *type*, a *size*, and a *value*. The *value* may bear different data structures according to each different *type*. In a session, the first command describes the formats of the remaining commands. The remaining commands may be inter-related.

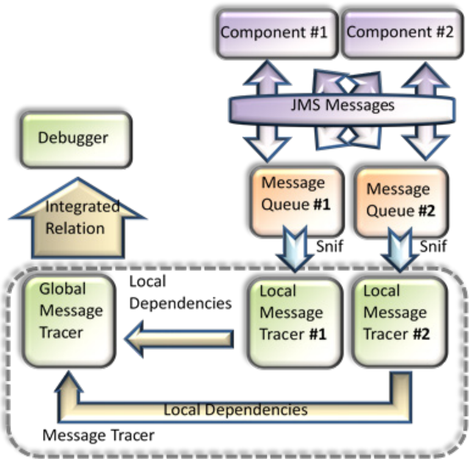


Figure 3. Message Tracer operation diagram

IV. APPLICATION DEPENDENCY TRACING FOR MESSAGE ORIENTED MIDDLEWARE

Existing application dependency tracing mechanisms are ineffective for distributed applications that employ MOM because MOM does not obey the synchronous threading model. We propose Message Tracer as shown in Figure 3 to approach the problem by observing the network traffic of message brokers and using domain knowledge of the message queue protocols to identify the dependencies of the messages around the message brokers. Message Tracer can trace dependency at per-message granularity. It does not rely on assumptions on the threading model of the message broker.

The operation of Message Tracer is presented in Figure 3. Message Tracer consists of a collection of *local tracers*, and one *global tracer*. For each message queue (message broker), a *local tracer* will be deployed to monitor its network traffic and identify the dependency of the messages of the message queue. Each *local tracer* will then submit the *local dependencies* to the *global tracer*, where *local dependencies* will be integrated into (global) *dependencies*. The *global tracer* also exposes an interface to be used by an external debugger for fetching the identified dependency information of the distributed application in question.

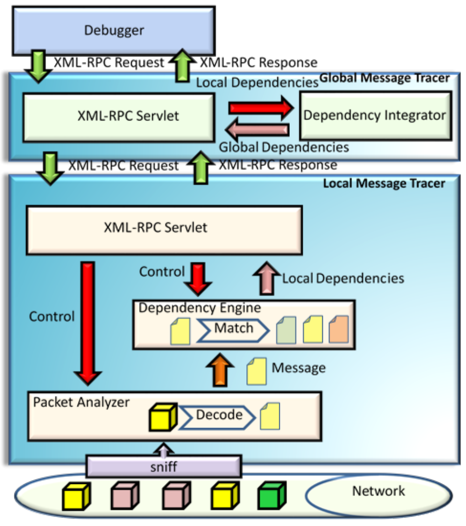


Figure 4. Message Tracer architecture

A. *Message Tracker Architecture*

The architecture of Message Tracer is presented in Figure 4. Each *local tracer* includes a packet analyzer, a dependency engine and an XML-RPC Servlet [16]. The *global tracer* includes a dependency integrator and also an XML-RPC Servlet. The dependence integrator integrates the local dependencies into global dependencies. The XML-RPC Servlets are used for communication and coordination of the tracers and the external debugger. Details about the dependency engine and the packet analyzer are given in Sec. IV.B and Sec. IV.C.

B. *Packet Analyzer*

To avoid intrusive modification to message broker software and to minimize performance overhead, Message Tracer uses packet sniffing to observe the network traffic of a message broker. The packet analyzer is responsible for sniffing the traffic around a message broker. The packet analyzer is also responsible for decoding the sniffed packet data into corresponding AMQP or OpenWire messages, which will later be used for dependency tracing. One challenge in the design of the packet analyzer is that packet sniffing is a best-effort process, and it may not capture all the packets of a message broker communication session. Those packets missed by the packet analyzer may eventually lead to inaccuracy in the dependency tracing. To address the issue, the packet analyzer incorporates a novel heuristic based on knowledge of message broker protocol structure to alleviate the impact of packet loss. In the following, we will introduce the design details of Packet Analyzer.

1) Packet Sniffing and Reassembly

The packet analyzer uses `jNetPcap` [17], a Java wrapper of the famous `libpcap` packet capturing library for packet sniffing. Both AMQP and OpenWire prefer using TCP as the transport protocol, so the packet analyzer can simply sniff the traffic only on those TCP ports used by the message brokers to reduce performance overhead and chance of packet loss.

Both AMQP and OpenWire are stateful protocols. The first few handshake messages for both protocols carry magic numbers that can be used to identify the protocols. The packet analyzer will detect presence of the magic numbers in the sniffed traffic to further reduce false positives (there might be spurious connections made to the message broker). The sniffed packets are reassembled into TCP segments by `jNetPcap` followed by the packet analyzer reintegrating the segments into corresponding layer-7 AMQP / OpenWire messages.

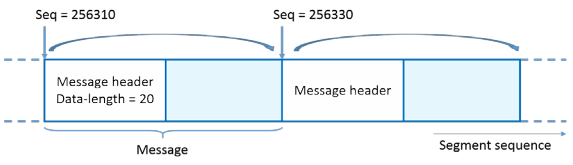


Figure 5. Message reintegration

2) Packet Loss and Message Reintegration

Because the message payload of both AMQP and OpenWire can be of variable length and the payload may include symbols that coincide with delimiter symbols, both protocols have a `size` field in the message headers to indicate the length of the messages. As shown in Figure 5, the layer-7 messages can then be seen as a chain of consecutive TCP segments. The packet analyzer will use the sequence number of a message and the size of the message to determine the segment number and the offset of the next message.

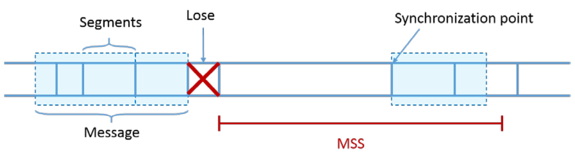


Figure 6. Recovery message chain

A challenge we faced in the design of the packet analyzer is that packet sniffing may miss some of the packets transmitted or received by a message broker. As a result, some of the messages may not be properly reintegrated by the packet analyzer. If a lost packet contains crucial information such as the message `size` field, the subsequent chain of messages may all be lost. Or, even worse, without further validation of the message format, this could easily lead to misinterpretation of subsequent messages and incorrect dependencies.

The packet analyzer adopts a heuristic to alleviate the impact of lost packets in packet sniffing. To recover from lost packets, we make the assumption that if a message is larger than TCP maximum segment size (MSS, which is usually 1500 bytes long), it must be split into several TCP segments of MSS length and a final TCP segment (with length < MSS length) that carry the residual message data. As shown in Figure 6, if a chain of messages is broken due to packet loss, the last segment with a length smaller than MSS must be the end of a message, and the next segment following the broken segment, must be the beginning of a new message.

3) Message Decoding

After message reintegration, the packet analyzer will decode the messages and extract the identifiers needed for dependency tracking (Sec. IV.C). The packet analyzer can decode the messages of OpenWire and AMQP 1.0.

For OpenWire, there are two key options that affect the message format, which have to be handled by the packet analyzer. The first is the tight encoding option, which indicates if a message should be compressed. The second option is the caching option, which indicates if long identifiers in an OpenWire message should be cached and referenced subsequently in a session by the cache IDs. The prototype implementation of packet analyzer utilizes codebase from ActiveMQ to decode OpenWire messages.

For AMQP 1.0, we chose to build our own decoder for AMQP 1.0 because the protocol is straightforward to implement. For the implementation, we had made references to the INETCO iAmqpDecode [18] AMQP 1.0 decoder written in C.

C. Dependency Tracing

The JMS standard requires the identifier field *message id* in the message header. JMS further requires that the *message id* for each message delivered through the same queue or topic has to be unique. For dependency tracing, Message Tracer first relies on the ip/port numbers to identify the queues or

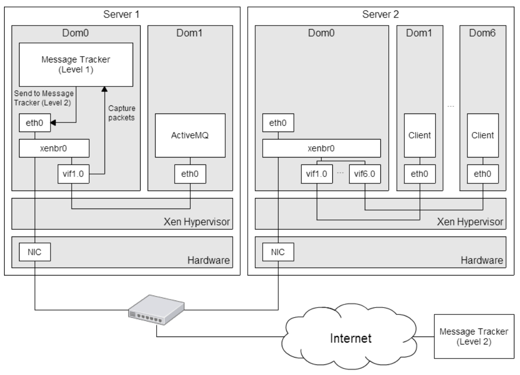


Figure 7. Testbed for experiments

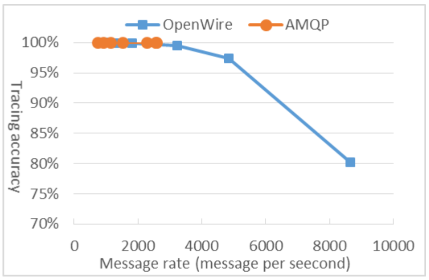


Figure 8. Message delay time and tracing accuracy

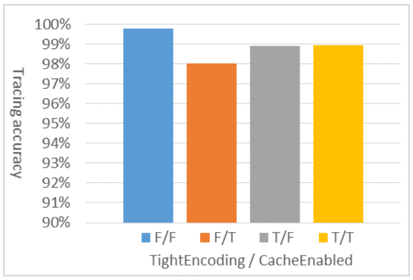


Figure 9. Tracing accuracy for different OpenWire options

relies on the topic names to identify the topics allocated on a message broker. After that, Message Tracer will use the *message ids* to establish the local dependencies of the messages for each queue and topic on the broker. For example, if a message *MA* is sent to topic *t* with message id *i* at message broker Z from client *A*. Later if Message Traccer observes a message *MB* with topic *t* and identifer *i* is being delivered to client *B* by broker *Z*, Message Tracer will then determine that there exists a local dependency from *MA* to *MB*.

V. Evaluation

We set up a testbed as shown in Figure 7 for the experiments. The testbed includes two physical servers and a bunch of virtual machines (VMs). Each server is equipped with 16 processros, 16 GB RAM, and 2 TB SATA disk storage. The servers are connnected to a 1 Gbps Ethernet network. Both servers use Fedora Core 16 x86_64 as the host OS with Xen hypervisor 4.1.4. The guest OS used by the VMs is also Fedora Core 16 x86_64. The message broker used in the testbed is Apache ActiveMQ 5.9. We used Tomcat 8.0 as the servlet container.

As shown in Figure 7, on Server 1, we set up a VM denoted as Dom1 for the ActiveMQ message broker. The Message Tracer prototype is deployed on Dom0 of Server 1. Message Tracer sniffed the network traffic of the ActiveMQ message broker and traced the message dependencies. On Server 2, we set up six VMs, each of which ran a flow generator. The flow generators were used to generate syntheic traffic to AcitiveMQ for the experimetns. Each flow generator can emulate a configurable number of clients that connect to the message broker. Each client may choose to use either OpenWire or AMQP as the message queue protocol. Each client can either act as a producer, a comsumer, or random roles. The number of messages to be generated and the time interval between consecutive messages are also configurable. For experiment purposes, each message generated by the flow geneators, is tagged with a universally unique identifier (UUID). Each flow generator maintains a log of sent / received messages. We rely on the logs and the UUIDs to establish the ground truths for the message dependencies.

A. Message Rate and Tracing Accuracy

In this experiment, we set up two clients, one as the producer and the other as the consumer. Both the producer and the consumer connected to the same queue at the message broker. In the experiment, the producer periodically pushed messages into the queue with a fixed time delay between message pushes. On the other hand, the consumer kept popping messages from the queue with the same fixed time delay. A total of 100,000 of messages were transmitted from the producer to the consumer for each round of the experiment. We used both AMQP and OpenWire protocols in the experiment. For OpenWire, we used tight encoding (TightEncoding = TRUE) and no caching (CacheEnabled = FALSE).

Time delay between message pushes (ms)	0	0.1	0.2	0.4	0.6	0.8	1
OpenWire Flow (MPS)	8651	4851	3223	1828	1276	987	802
AMQP Flow (MPS)	2587	2552	2268	1526	1148	906	743

Table 1. Time delay and effective message rate

The parameters used in this experiment is shown in Table 1, and the result is presented in Figure 8. For both message queue protocols, Message Tracer was able to achieve almost 100% accuracy for message rate below 3000 message per second (MPS). The accuracy drops to 80% for OpenWire at the message rate of 8,651 MPS. We also noticed the maximum message rates that can be supported by our testbed are different between AMQP and OpenWire. Our testbed can achieve significantly higher message rate for OpenWire. This is possibly due to variations in the implementation of the two protocols in ActiveMQ and the flow generator.

In this experiment, we also looked at the effect of using tight encoding and caching in OpenWire on the tracing accuracy. The flow generator is set to generate messages at 0.1 ms time delay. We then toggled the `TightEncodeing` option between `TRUE` (T) and `FALSE` (F) and also toggled the `CacheEnabled` option between `TRUE` (T) and `FALSE` (F), and observed the resulting tracing accuracy. As shown in Figure 9, the effect of both the options on the tracing accuracy is negligible as the difference is less than 2%.

B. Concurrent Clients and Tracing Accuracy

In this experiment, we looked at the effect of concurrent client connections on the tracing accuracy. We set up 4 VMs. Each VM runs a given number of clients that connect to the message broker. We used the publish/subscribe model for this experiment, and there are 4 topics set for this experiment. Each client randomly picks one of the 4 topics to subscribe to. Each client will then repeatedly publish new messages to the 4 topics at random.

We vary the number of concurrent clients on each VM and measure the tracing accuracy for each case. The result is presented in Figure 10. The corresponding message rate for

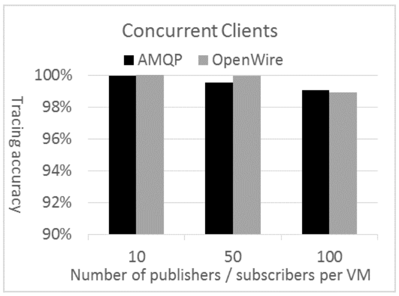


Figure 10. accuracy of concurrent clients with AMQP and OpenWire

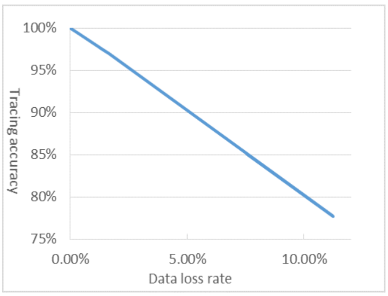


Figure 11. Tracing accuracy and data loss rate

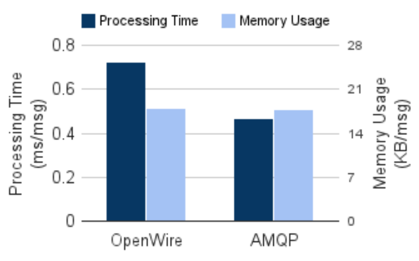


Figure 12. Analysis time and peak memory usage with OpenWire and AMQP

each setting is shown in Table 2. We can see the number of concurrent clients does not affect the tracing accuracy (the tracing accuracies are all above 99%). In retrospect to the result of Sec. V.B, we believe the minor fluctuation in the tracing accuracy here is mainly due to the difference of the message rate in each settings.

# of publishers / subscribers per VM	10/10	20/20	40/40
OpenWire (MPS)	3702	3868	4202
AMQP (MPS)	3766	3787	4148

Table 2. Message rate (message per second) for different concurrency settings

C. *Impact of Packet Loss*

To achieve low performance overhead, Message Tracer uses packet sniffing to observe the network traffic of message brokers. Packet sniffing is a best-effort process and occasional packet loss can be expected. On the other hand, AMQP and OpenWire are both stateful protocols, so packet loss is likely to have profound impact on the tracing accuracy. We set up an experiment to evaluate the impact of the packet loss in sniffing on the tracing accuracy. The result is presented in Figure 11. We use OpenWire as the message queue protocol for this experiment. Overall, the tracing accuracy drops linearly with increasing data loss rate. As expected, the impact is indeed noticeable.

D. *Performance Overhead of Message Tracker*

Figure 12 shows the performance overhead of Message Tracer with respect to the experiment settings of Sec. V.B. The time delay is 1 ms. For both OpenWire and AMQP, it requires 18KB memory space to maintain the record in Message Tracer for each message delivered through the message broker. Each record includes the message ID, the client IP address, and the client port number.

The processing time for each message is also presented in Figure 11. The processing time of OpenWire is higher than the processing time of AMQP. This is because Message Tracer uses the built-in ActiveMQ decoder to decode OpenWire packets, while Message Tracer employs a custom-built AMQP decoder. The custom-built AMQP decoder is more efficient than the ActiveMQ OpenWire decoder, because it only decodes the portion of information of relevance to dependency tracing

VI. CONCLUSION

We present Message Tracer to trace the message dependencies in distributed applications that are built on top of message oriented middleware. Message Tracer uses packet sniffing to observe the network traffic of message brokers and relies on knowledge of the message broker protocols to establish the message dependencies. Message Tracker incurs minimal overhead. It does not require modification to the application software or the message broker software. Packet sniffing may suffer packet loss. Message Tracker employs a novel heuristic to address the impact of packet loss in the message reintegration process. Overall, Message Tracer was able to achieve a very high (>95%) accuracy in tracing the dependencies of application messages for most cases. Even in the worst case, where the message broker is stressed to its peak throughput, the tracing accuracy was still well above 75%.

ACKNOWLEDGEMENT

This study is suported by the Cloud Computing Center for Mobile Application of the Industrial Technology Research Institute. The study is also partially supported by the Ministry of Science and Technology of the Republic of China under grant number 101-2221-E-009-076-MY3.

REFERENCES

[1] X. Chen, M. Zhang, Z. M. Mao, and P. Bahl, "Automating network application dependency discovery: experiences, limitations, and new solutions," presented at the USENIX Operating Systems Design and Implementation, San Diego, California, 2008.

[2] R. Fonseca, G. Porter, R. H. Katz, S. Shenker, and I. Stoica, "X-trace: a pervasive network tracing framework," presented at the USENIX Conference on Networked systems design implementation, Cambridge, MA, 2007.

[3] B. H. Sigelman, L. A. Barroso, M. Burrows, P. Stephenson, M. Plakal, D. Beaver, et al., "Dapper, a large-scale distributed systems tracing infrastructure," Google research, 2010.

[4] B. C. Tak, C. Tang, C. Zhang, S. Govindan, B. Urgaonkar, and R. N. Chang, "vPath: precise discovery of request processing paths from black-box observations of thread and network activities," presented at the Proceedings of the 2009 conference on USENIX Annual technical conference, San Diego, California, 2009.

[5] M. Y. Chen, E. Kiciman, E. Fratkin, A. Fox, and E. Brewer, "Pinpoint: Problem Determination in Large, Dynamic Internet Services," presented at the Proceedings of the 2002 International Conference on Dependable Systems and Networks, 2002.

[6] E. Thereska, B. Salmon, J. Strunk, M. Wachs, M. Abd-El-Malek, J. Lopez, et al., "Stardust: tracking activity in a distributed storage system," presented at the Proceedings of the joint international conference on Measurement and modeling of computer systems, Saint Malo, France, 2006.

[7] M. Y. Chen, A. Accardi, E. Kiciman, J. Lloyd, D. Patterson, A. Fox, et al., "Path-based faliure and evolution management," presented at the Proceedings of the 1st conference on Symposium on Networked Systems Design and Implementation - Volume 1, San Francisco, California, 2004.

[8] P. Reynolds, C. Killian, J. L. Wiener, J. C. Mogul, M. A. Shah, and A. Vahdat, "Pip: detecting the unexpected in distributed systems," presented at the Proceedings of the 3rd conference on Networked Systems Design & Implementation - Volume 3, San Jose, CA, 2006.

[9] E. Curry, "Message-oriented middleware," Middleware for communications, pp. 1–28, 2004 2004.

[10] IBM.com. WebSphere MQ.

[11] Apache.org. (4-22). ActiveMQ. Available: http://activemq.apache.org/

[12] Pivotal Software. RabbitMQ. Available: http://www.rabbitmq.com/

[13] D. E. Mark Hapner, S. E. Rich Burridge, S. S. E. Rahul Sharma, S. S. E. Joseph Fialli, S. S. E. Kate Stout, and I. Sun Microsystems, "Java Message Service API," ed. https://jcp.org/aboutJava/communityprocess/final/jsr914/index.html: Oracle.

[14] S. Vinoski, "Advanced Message Queuing Protocol," IEEE Internet Computing, vol. 10, pp. 87–89, November 2006 2006.

[15] Apache.org. OpenWire.

[16] S. Allen, J. Lapp, and P. Merrick, XML remote procedure call (XML-RPC), 2006.

[17] Sly Technologies. jNetPcap.

[18] INETCO.com. iAmqpDecode.