Extending ARM to Support n-D Correlation

Akhil Sahai, Jinsong Ouyang, Vijay Machiraju
Software Technology Laboratory
HP Laboratories Palo Alto
HPL-2000-169
December 11th, 2000*

ARM, E-service, management, optimization, correlation

ARM [1] and other related transaction measurement techniques [2,4] are aimed at providing a mechanism for measuring transactional parameters and for relating the measurements of one transaction with those of another. A "correlator" passed from one transaction (referred to as the parent) to another (child) is used to establish a parent-child relationship between the two transactions. However, each of these techniques is limited to having a single (or no) parent for every sub-transaction. In managing distributed applications, a case often arises that warrants correlating a single child with multiple parents. In other words, a transaction could be viewed as a component of two or more parent transactions. In this paper, we extend state of the art in transaction measurement from unidimensional correlators to multi-dimensional correlators.

* Internal Accession Date Only
Approved for External Publication

Extending ARM to Support n-D Correlation

Akhil Sahai, Jinsong Ouyang, Vijay Machiraju
E-Services Software Research Department
HP Laboratories, 1501 Page Mill Road, Palo-Alto, CA 94034

Abstract: ARM [1] and other related transaction measurement techniques [2,4] are aimed at providing a mechanism for measuring transactional parameters and for relating the measurements of one transaction with those of another. A "correlator" passed from one transaction (referred to as the parent) to another (child) is used to establish a parent-child relationship between the two transactions. However, each of these techniques is limited to having a single (or no) parent for every sub-transaction. In managing distributed applications, a case often arises that warrants correlating a single child with multiple parents. In other words, a transaction could be viewed as a component of two or more parent transactions. In this paper, we extend state of the art in transaction measurement from uni-dimensional correlators to multi-dimensional correlators.

A. INTRODUCTION

Transaction measurement is an important aspect of application management. The ARM API [1] is a simple API that applications can use to pass vital information about their transactions. A transaction is loosely defined as any unit of work within the application logic that can be marked with a "start" and "stop". Examples of transactions include:

1. body of a function or method (start at the beginning of the function and stop at the end of the function)
2. a critical section of code (start at the beginning of the critical section and stop and the end of the critical section)
3. the lifetime of an object (start in the constructor of the object and stop in the destructor)
4. a "purchase order" transaction (start when the user initiates a purchase order and stop when the user is given a confirmation number)
5. A "book buying" experience (start when the user logins in to a web site and stop when the book is shipped to or received by the user).

The exact number and nature of transactions are left to the discretion of the application developer. From the above examples, it can be noted that transactions can represent a contiguous piece of code (e.g., 1, 2) or functionality spread across several methods or components (e.g., 3, 4, 5). They can be used to represent system/application logic (e.g., 1, 2, 3) or business logic (e.g., 4, 5). Further, transactions can be short-term (execution completes within seconds or minutes) or long-term (execution takes days or months to complete). However, in all these cases, there should be a well-defined "start" and "stop" that could be inserted into the application logic to mark the boundaries of the transaction.

In addition to measuring transactions in isolation, it is often beneficial to relate transactions with each other. The parent-child relationship ties a parent transaction to its sub-transactions. This can be used to measure the parent transaction as a whole and to relate that measure with the measures of each of the sub-transactions. A common use of this is to define a parent transaction as the client-initiated transaction and to define sub-transactions for portions of the parent transaction that execute within server components (Figure 1). For instance, a purchase order transaction initiated in a browser in an e-commerce application could be defined as the parent transaction while the resulting execution in a web server could be defined as the sub-transaction.

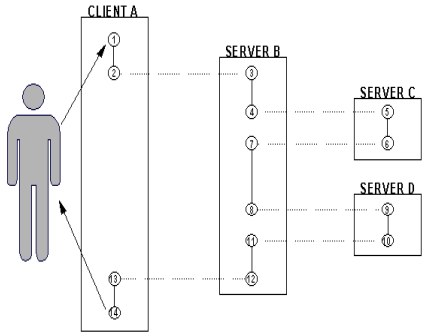

Figure 1: Transactions and sub-transactions

The parent-child relationship is established by passing a correlator from the parent transaction to the child transaction. A correlator is a unique identifier that can be used to distinguish one instance of a transaction from another. In the example above, the browser passes a correlator to the web server.

While applying these techniques to relate transactions in distributed components, we found that the ability provided by ARM to relate a sub-transaction to a single parent was inadequate. In many cases, a single piece of functionality participated in two or more logical transactions at the same time. In this paper we explain a simple extension to existing correlation mechanisms to handle what we call as n-D correlation. The rest of this paper is organized as follows: Section B shows some

examples that demonstrate the need for n-D correlation. In section C, we suggest a solution to handle n-D correlation. We conclude with some thoughts in section D.

B. NEED FOR n-D CORRELATION

Consider the following example, which commonly arises in many object pool applications (e.g., web server farm, database connection pool, thread pool, etc).

Example 1: Object Pool

An object pool is a set of objects – all of which belong to the same type. Consider an object pool of objects O_1, O_2, … O_n. Let us assume that each object supports a method M. The implementation of M constitutes a sub-transaction that can be though of as participating in two logical transactions, as demonstrated in Figure 2 – (1) a logical transaction (P1) spanning the lifetime of the object and (2) a logical transaction initiated by the client that invokes these methods (P2). It would be useful to understand how many times M was called within one object in the pool versus another. Hence, relating M1 with the lifetime of the object that it is part of is necessary. On the other hand, it is also useful to understand the contribution of M to the overall response time of the client's invocation. Hence, relating M with the client invocation is necessary.

In this example, the sub-transaction that corresponds to M should be correlated along two dimensions that correspond to the two logical transactions.

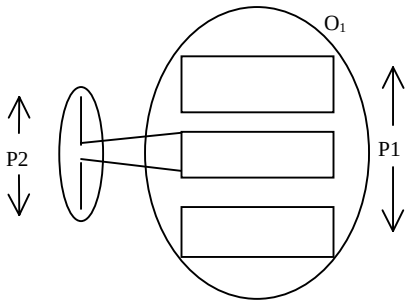

Figure 2: An object in the object pool. Method M is part of two logical transactions – P1 and P2.

Example 2: Event-loop

Event loops are commonly used in many distributed systems and event-driven systems. An event loop stays in an infinite loop waiting for new events. Whenever a new event is received it is handled before waiting for more events (Figure 3). Event loops are usually implemented using queues or multiple threads.

Consider the example of an event loop shown in Figure 3. The portion of the code marked as T is a sub-transaction that executes logically within two parent transactions – P1 that encompasses the loop and P2 that constitutes the transaction of the event generator. This is another example where T should be correlated along two dimensions – P1 and P2.

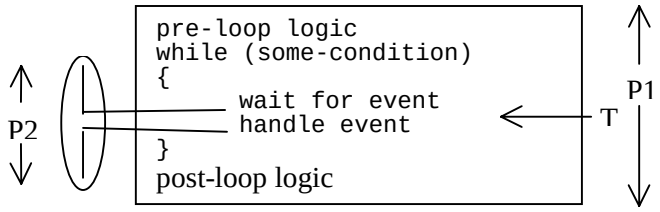

Figure 3: The event handler constitutes a transaction T, which is part of two logical transactions – P1 and P2.

Example 3: Data flow computation

To understand the need for correlating across more than two dimensions, consider the data flow computation shown in Figure 4. The data flow computation is characterized by the fact that the computation is triggered only when all the inputs are available. In the example shown in Figure 4, the data flow computation module (DFM) has n inputs I_1, I_2, …, I_n. Let us assume that each of these inputs is associated with a transaction (P_1, P_2, … P_n). In other words, P1 is a transaction that is defined by a module (not shown in the figure) as starting at some point before sending the input I_1 to DFM and ending at some point after sending the input I_1 to DFM. If we demote the work performed by DFM as transaction T, we can think of T as a sub-transaction of n parents P_1, P_2, … P_n.

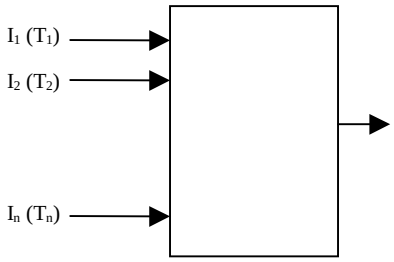

Figure 4: The computation of DFM is a sub-transaction of multiple parent transactions, each associated with one input to the DFM.

The need for n-D correlation arises from the fact that logical transactions can be arbitrarily defined based on business or other requirements. It is quite possible that

two logical transactions intersect at or share the same functionality. Existing approaches and APIs are limited in their ability to handle these scenarios. For example, handling these scenarios in ARM would require making multiple calls to arm_start() to indicate the start of the same sub-transaction which is not desirable for several reasons:

1. Each call to arm_start() results in a new handle being generated effectively creating a new sub-transaction.
2. Even if one could pass the same handle to arm_start(), the semantics of ARM do not state that one could do so and hence the ARM implementation library may choose to ignore multiple starts with the same correlator.
3. Using multiple starts is logically incorrect since we are starting a single sub-transaction, not multiple sub-transactions. As a consequence, multiple starts with the same handles may result in multiple response times being associated with the same sub-transaction.

So, we propose an extension to ARM or ARM-like mechanisms to deal with this problem. The extension comes in two parts:

- Extension to the instrumentation API, and
- Extension to the handlers for the instrumentation API.

In the case of ARM, these extensions map directly as extension to the ARM API and extension to the ARM library.

C. HANDLING n-D CORRELATION

In the ARM API, a correlator is specified as part of the arm_start() method.

```
start_handle = arm_start(
      tran_id, flags, data, data_size
)
```

In this method, tran_id represents the transaction type identifier and start_handle represents the transaction instance identifier returned by the library. You can notice that the correlator is not directly specified as a parameter to the method. Instead it is embedded in the "data" attribute using an encoding mechanism specified by the ARM standard. Specific bits within the "flags" attribute are used to indicate whether a correlator is encoded in the "data" attribute or not.

This scheme should be changed to allow applications to specify more than one parent transaction correlator. To retain backward compatibility, one could design a new encoding mechanism for the "flags" and "data" attributes to accommodate this. Instead of going deep into the encoding mechanisms and in order to support other ARM-like transaction measurement schemes, we just propose a conceptual extension to this API. Physically, this extension could be realized by creating similar API calls or new encoding mechanisms as in the case of ARM. The conceptual extension is given below:

```
start_handle = start(
      tranid, parent_tran_handles[],
      other_fields
)
```

In case, an application does not use separate calls to start and stop transactions, but intends to send complete transaction information within one single call, it can use the following variant of ARM API for transaction completion:

```
complete(
    own_tran_handle, tranid,
    parent_tran_handles[],
    tran_status, other fields
)
```

In both these extensions, tranid represents the transaction type identifier for the new transaction being marked. In the first extension above, start_handle represents the new transaction instance identifier returned by library. In the second extension, own_start_handle is an instance identifier provided by the application itself to the library.

The array of parent_tran_handles is used to represent correlators corresponding to all the parent transactions.

We now show how one could perform n-dimensional correlation for the example of Object Pool shown in Figure 2. Other examples in section C can be handled similarly.

The client in Figure 2 passes its handle P2 as the correlator to M. The mechanisms used for passing correlators between two components are beyond the scope of this discussion. M also has access to P1, which is the start handle generated when the constructor of the object marked the beginning of its transaction. M can now make a call to start() using both these correlators as shown below:

```
T = start(
      M, {P1, P2}, other_fields
)
```

An API handler that implements this call can store the relationship between T, P1, and P2 in its internal data structures.

D. CONCLUSION

ARM provides for uni-dimensional correlation between transactions. However, in many situations there are multiple dimensions to a transaction. To account for this, we extended the API to be able to specify multiple parent correlators.

E. ACKNOWLEDGEMENT

We would like to thank Aad Van Moorsel for his suggestions and feedback on the paper.

F. REFERENCES

[1] ARM Working group, Application Response Measurement API guide
http://www.omg.org/regions/cmgarmw/index.html

[2] Evans, J. Klein, and J. Lyon, *Transaction Internet Protocol –Requirements and Supplemental Information, 1998*
http://www.landfield.com/rfcs/rfc2372.html

[3] J. T. Park and J. W. Baek. *Web-based Internet/Intranet service management with QoS support*. IEICE Trans. Communations., e82-b:11, 1999.

[4] Vantage Point Web Transaction Observer
http://www.openview.hp.com/products/webtransobserver/