Monitoring and Diagnosing Applications with ARM 4.0

Mark W. Johnson
IBM Corporation

The ARM (Application Response Measurement) standard provides a way to manage business transactions. By embedding simple calls to an agent supporting ARM an application can be managed for availability, service level agreements, and capacity planning. ARM provides a way for applications to provide additional information about transactions, including the relationship between parent and child transactions end-to-end. This paper describes the API, and then explores how ARM information can be used to monitor and diagnose application response time.

Introduction

During their widespread introduction in the 1960s and 1970s, computer systems had a dramatic impact on many businesses. The deployment of personal computers in the 1980s had a similar impact, providing benefits to small organizations that had benefited only indirectly from the earlier computing technology. Beginning with the explosion in usage of the Internet in the mid-1990s, the greatest advantages are realized by organizations that exploit network computing. They provide new applications that were unthinkable even a few years ago by combining handheld devices, PCs, powerful shared computers, large databases, and networks that connect them.

These applications provide unprecedented opportunities for organizations to reach more customers with ever more useful services. They boost productivity and increase the flexibility and responsiveness of the organizations that use them. Because they are so important, these applications, and the networking and computing systems that they run on, are critical to the success of these organizations.

These applications are fundamentally different than their predecessors. They have more dependencies on more systems spread over an ever wider geographical area. They partition function throughout the network, and they exploit many different technologies. It is not possible to manage just one technology or one system and understand what the business needs to know. Are the applications doing what they are intended to do? Are transactions completing successfully? Are response times satisfactory? Where are there bottlenecks? In many cases, information from the application itself is needed to answer these questions.

The ARM (Application Response Measurement) standard defines a way for applications to provide information to help answer these questions. The applications provide critical information about business transactions from the perspective of the business operations rather than, say, the network. The applications indicate how transactions are related to business transactions initiated by end users. With this information, application management software can measure and report service level agreements, get early warning of poor performance, notify operators or automation routines immediately if transactions are failing, and help isolate where slowdowns are occurring.

ARM standard

Figure 1 shows how business applications are managed with ARM. ARM is a simple API (application programming interface) that applications or middleware use to pass vital information about transactions to management software. A transaction is any unit of work that the developer of an application considers significant. Examples are interactive transactions in which the user clicks a mouse button and is waiting for a response, remote calls to a database server, and batch jobs.

The application calls ARM just before a transaction starts and then again just after it ends. The ARM calls identify the application, the transaction, its completion status, and optional context data about the transaction,

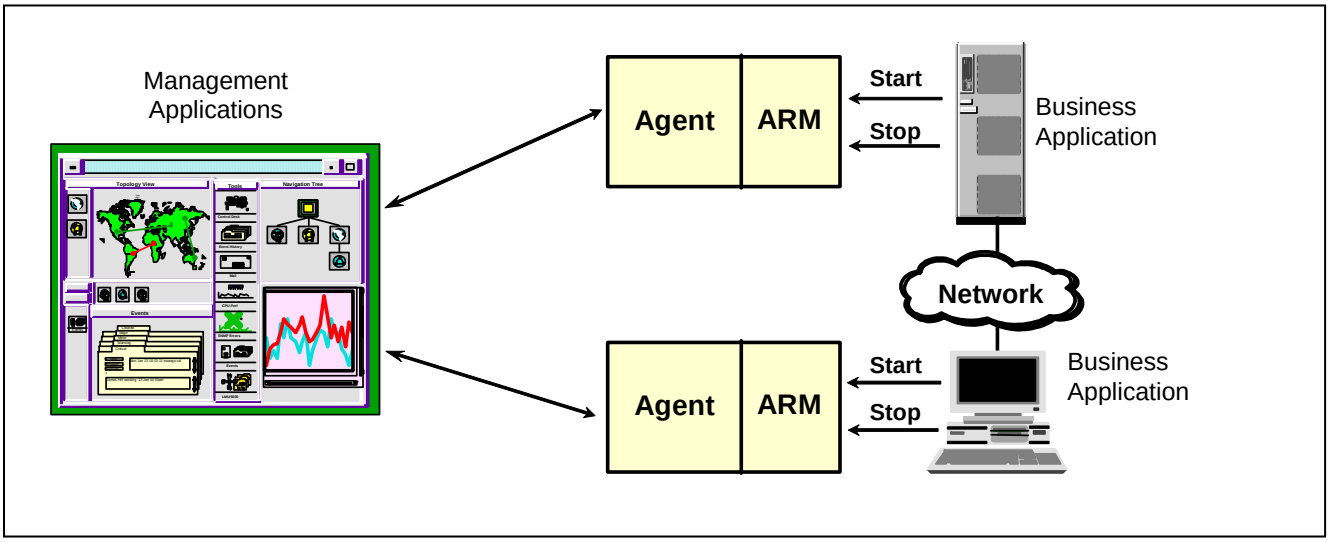

Figure 1 ARM Overview

such as the user who initiated it and the parent/child relationship between transactions. An agent that implements ARM receives the program calls. Based on the calls, it measures and monitors the transactions and makes information about them available to management applications.

Questions ARM helps answer

Information collected by ARM is sufficient for management solutions to accurately answer important questions without making guesses. Eliminating this guesswork helps both the users of the applications and the administrators responsible for making sure they are meeting the requirements of the business.

Is a transaction (and the application) hung, or are the transactions failing? Rather than waiting for phone calls from dissatisfied users, operators can get immediate notification when a transaction is not completing satisfactorily (if the network is available), and appropriate steps taken to remedy the situation. Alternatively, this information can be logged and accurate counts tracked over time.

Where is a transaction failing? A trace can be initiated to pinpoint the source of failures. Further analysis could indicate the specific cause, such as a held lock that is blocking a thread.

What is the response time? Having satisfactory response times is second in importance only to knowing whether a transaction is hung or is failing. In fact, from the perspective of the user, the only two things that really matter are whether it is working and how long it is taking. Response times are important elements in service level agreements.

Who uses the application, and how many of which transactions are used? Because the application calls the ARM API for each transaction, and (optionally) each user is identified, an accurate understanding of actual workload is obtained. Incorrect assumptions about workload are a big contributor to inadequate capacity planning. This same information can be used for charge-back accounting.

Where are the bottlenecks? If response times are unsatisfactory, where is the time being spent? By measuring child transactions in addition to the main transactions visible to the user, a lot of insight into where the most time is being spent can be learned.

How can the application or environment be tuned to perform better? By understanding which child transactions contribute the most to the total response time, administrators can take steps to minimize the execution paths that are taking the longest. A faster communications link might be installed, a server might be moved onto a network segment closer to its heaviest users, a larger system might be installed, or an application could have a few critical sections redesigned.

```
/* Initialization when application starts. */
arm_register_application(Order_App);                          /* Describes application type */
arm_register_transaction(Order_App, Query_Status_Tran);       /* Describe transaction type(s) */
arm_start_application   (Order_App_Instance);                 /* Once per running application instance */

/* Main loop: process input requests until we are terminated */
while(work_to_process)
{
    /* Process one transaction. Use arm_start_transaction() to start measuring transaction */
    get_caller_request   (Calling_Parent_ID, input_parameters);
    arm_start_transaction(Order_App_Instance, Query_Status_Tran_Instance, Calling_Parent_ID, my_ID);
    arm_bind_thread      (Query_Status_Tran_Instance); /* only needed if multiple threads are used */
    process_transaction  (input_parameters);           /* business logic */

    /* External function invocation; Use block/unblock to show how much we are externally blocked */
    arm_block_transaction  (Query_Status_Tran_Instance, Blocking_Condition_ID);    /* optional */
    invoke_external_service(my_ID, request_parameters);                           /* external call */
    arm_unblock_transaction(Blocking_Condition_ID);                               /* optional */

    /* transaction complete
    process_returned_data();                                        /* more business logic */
    return_response_to_caller(output_response);                     /* complete our caller's request */
    arm_unbind_thread     (Query_Status_Tran_Instance);             /* often not needed */
    arm_stop_transaction  (Query_Status_Tran_Instance, STATUS_GOOD); /* captures measurements */
}
/* Termination. Tell ARM that we're done. */
arm_stop_application   (Order_App_Instance);
```

Figure 2 Sample ARM Instrumentation in a Psuedo-Language

History of ARM

What was announced as the ARM API in June 1996 started as separate and independent projects at IBM and Hewlett-Packard. Both projects had similar goals, and each had resulted in implementations that were available as products. Both companies wanted to better serve the needs of the computing community by merging their independent efforts to create a vendor-neutral API using the C programming language. This has encouraged use of the ARM API in applications, and stimulated cross-industry and cross-platform support.

The technical work took the best ideas from each API, and molded them into a joint proposal which we now know as ARM. As the work progressed towards announcement, many users and vendors were consulted, so that by the time ARM was announced in June 1996 there was a long list of companies supporting the activity.

The ARM Working Group, a consortium of companies with varied interests in ARM, developed extensions to the API. These extensions were named "ARM 2.0" and were announced and made available in an updated Software Developers Kit in 1997. ARM 2.0 was adopted with minimal changes as a formal standard by The Open Group in 1998 [ARM98].

The Open Group adopted Java™ bindings as "ARM 3.0" in 2001 [ARM01]. The ARM 2.0 C and ARM 3.0 Java bindings are similar but not entirely compatible.

In 2003 The Open Group adopted both C and Java bindings in ARM 4.0 [ARMC03, ARMJ03]. In ARM 4.0 the C and Java bindings are compatible with each other. ARM 4.0 added extensions to enable more accurate timings in some environments, to make the semantics of application and transaction identity more flexible, and to increase the amount of context data about each transaction instance, such as the binding of a transaction to a thread.

ARM adoption in commercial applications

The use of ARM has steadily increased over the past several years. Most of the early adopters of ARM were companies that developed applications in-house. They instrumented the applications with ARM in order to improve the manageability of these applications. More recently, commercial applications have instrumented with ARM. Examples that are currently available include Siebel™ 7.7, SAS® 8.2, IBM® WebSphere Application Server™, and IBM DB2 Universal

Database™. Plug-ins have also been written for widely used components such as the Apache™ HTTP™ Server and Microsoft® Internet Information Services [EWLM04].

Commercial implementations of ARM, backed up by management applications, have been available for several years from several vendors. These are continuing to evolve to add new function and to support newer versions of ARM. These products receive and process ARM calls. In some cases they also provide instrumentation on behalf of applications, particularly in J2EE™ (Java 2 Platform, Enterprise Edition) environments. In these cases the application's EJB™ (Enterprise JavaBeans™) programs and servlets are measured without the application source code being changed.

Combining the instrumentation and ARM support in these commercial products, it is now possible to purchase solutions that trace web transactions end-to-end from the HTTP server, through multiple application server tiers, to database servers. Each transaction on these tiers is measured for status and response time, and the data put together for purposes of automated discovery, fault isolation, workload balancing, and capacity planning.

Description of the API

Figure 2 is an example in a psuedo-language showing the API calls. There are a few other function calls for specialized uses. The Java binding specifies methods that are equivalent. Complete descriptions can be found in the C and Java ARM specifications, along with working examples.

Using ARM

There are three steps to monitoring application performance with the ARM API.

In the first step the application architect selects the transactions to measure and the properties to describe them. This is the most important step. The architect considers who needs what kind of data, and what the data will be used for. It is common and useful for this process to be a joint collaboration between the users and developers of an application, and system and network administrators. A typical approach is to start with transactions that are visible to users or that represent major business operations, and then to augment these with transactions that are dependent on external services, such as a database operation, a RPC (Remote Procedure Call), or a remote queue operation. These will generally be child transactions of a user/business transaction. Knowing how these types of transactions are performing can be invaluable when analyzing problems, tuning applications, and reconfiguring systems and networks.

In the second step a developer modifies the program to include calls to the ARM API. The stub libraries in the developers' toolkit can be used for initial testing.

In the third step a system administrator replaces the stub libraries from the developer toolkit with an ARM-compliant agent and associated management applications. These may be commercially available or developed in-house. The distributed applications will now be monitored in ways that previously were not possible.

Selecting transactions to measure

Selecting the transactions to measure is one of the two most important decisions made by an architect designing the instrumentation strategy for an application. (The other key decision is selecting properties that describe the transactions). The selection is important because ARM agents and management applications depend on these properties to organize measurements for reporting and analysis.

Query Type	Quantity	Response Time Elapsed	Response Time Mean
Individual	500	1500	3.00
Business Payable	300	2550	8.50
Business Receivable	200	3000	15.00
TOTAL	1000	7050	7.05

Figure 3 Example Response Time Distribution

For example, consider an ARM agent that measures transactions to create service level reports of the mean response time over time intervals of an hour. If an application registers one transaction named "Query Account" and executes 1000 instances of it over an hour, the application calls arm_start_transaction() and arm_stop_transaction() 1000 times over the hour. All

1000 response time measurements are averaged to yield one mean. This may, of course, make perfect sense. However, consider the case where there are three types of accounts, one for individuals, one for business partner payable accounts, and one for business partner receivable accounts. The query for each account type might return significantly different amounts of data with corresponding differences in the complexity and response time of each query. Hypothetical values for an interval are shown in Figure 3.

The mean of all 1000 transactions is 7.05 seconds, but this value is not accurate for any of the three query types. Further, without the breakdown by query type, a capacity analyst would not arrive at correct projections if one of the three types is projected to increase significantly. In this situation, modeling three separate transactions is much more useful.

A good guideline for instrumentation is to use identity and context properties to distinguish transactions from each other if the response time and/or workload for one group differ significantly from another group.

Describing transactions

Each application and transaction is associated with up to four types of properties.

Identity properties define the type or class of an application or transaction. All instances of a given type or class have the exact same identity properties.

Context properties are additional properties that may be unique for each instance of an application or transaction. They may be important for three reasons. First, they may provide additional information about the business context, such as the name of the user or a business process ID (Identifier). Second, they may provide more granular information about a transaction, such as the part number in a transaction that updates an inventory database. Third, they may provide information about the environment in which a transaction is executing, such as the identity of the process or thread in which the transaction executes.

Relationship properties show how transactions are related to each other. The most common transaction relationship is when one transaction is the parent of another transaction (a classic client/server transaction). The child transaction may or may not complete before the parent transaction can complete, and the parent transaction may or may not suspend until the child transaction completes. ARM currently models these relationships as a tree with each transaction having zero or one parent. In the future ARM may be extended to model transactions as an acyclic directed graph in which each transaction may have multiple parents. Graphs are useful for modeling workflow transactions with parallel paths that converge at synchronization points.

Measurement properties are the measurements that are analyzed to locate failures, isolate bottlenecks, and to understand a transaction's performance. The core measurements are status, response time, time of day, and blocked time. There may also be related useful information, such as the number of bytes in a file transfer or the number of records in a data backup job. There may also be measurements about the environment in which a transaction is executing, such as the number of threads or buffers allocated, or the number of transactions queued up for processing. In general, any factors that may substantially affect the response time of a transaction are good candidates to use.

Using identity vs. context properties

The instrumentation architect has two options for differentiating the account type queries.

1. By identity properties. The architect could name each transaction the same way, such as "Query Account", and use a (name, value) **identity** property, such as ("Type", "Individual"). Alternately, the architect could achieve the same result by using a different transaction name for each query type, such as "Query Individual Account".

2. By context properties. The architect could name each transaction the same way, such as "Query Account", and use a (name, value) **context** property, such as ("Type", "Individual").

Selecting identity vs. context properties is influenced primarily by the number of unique transactions that would need to be registered. A good example is an application that will provide to ARM the URI (universal resource identifier) for each transaction. If the URI contains parameters, there could be very many unique URIs. It would be prohibitively expensive to take the time to determine if each URI had been previously registered, register it with arm_register_transaction() if it is new, and then execute the transaction with the newly

Category	Property Name	Description	Application	Transaction
Identity	Name	Unique descriptive name.	**Mandatory**	**Mandatory**
	(name,value)	Any (name,value) pairs that along with the 'Name" help provide a unique identity.	Optional	Optional
Context	User	The name or ID of the user associated with this transaction.	N/A	Optional
	Thread	The thread(s) executing a transaction	N/A	**Recommended**
	(name,value)	Any (name,value) pairs that help refine the understanding of the application/transaction.	Optional	Optional
	Group	Names a group, such as a logical cluster, to which an application instance belongs.	Optional	N/A
	Instance	A distinguishing identifier for an application instance, such as a process ID	Optional	N/A
Relationship	Current correlator	A byte array ID, representing the current transaction instance, used to correlate transactions end-to-ed across a calling chain.	N/A	**Recommended**
	Parent correlator	A byte array ID, representing the "parent" transaction instance, used to correlate transactions end-to-ed across a calling chain.	N/A	**Recommended**
Measurement	Status	The status of the completed transaction.	N/A	**Mandatory**
	Response time	The response time of the completed transaction.	N/A	***Measured by ARM***
	Time of day	The time of day that a transaction executed.	N/A	***Measured by ARM***
	Blocked time	The amount of time that a transaction is blocked waiting for an external event.	N/A	**Recommended**
	"size"	A measurement that indicates the 'size' of a transaction, such as the number of bytes in a file transfer or the number of files backed up.	N/A	Optional
	other	Measurements related to a transaction that would be useful when analyzing response time.	N/A	Optional

Figure 4 Application and Transaction Descriptive Properties

assigned transaction ID. Providing the URI as a context property would be a suitable design approach.

On the other hand, if many instances of a transaction are described by the same property values, there are performance advantages to using identity properties. Because the identity properties are provided once when the transaction is registered, the agent can perform all classification at that time, and no further parsing or analyzing of the properties is needed as each instance executes. If they are context properties, they must be parsed and analyzed each time the transaction executes.

Understanding relationship properties

In the client/server model, a program providing a service (the server) will make that service available to other programs through a programming interface. Another program (the client) issues requests to the server to provide the service. In order to perform the requested function, the server may play the role of a client to other servers by issuing requests to them. In each case the server processes the request and returns status to the client. Figure 5 is an example of nested transactions.

A user requests some action at Client A via the mouse or keyboard (1). Client A processes the transaction until it needs a service from Server B, so it sends a request (2). Server B receives the request (3), processes for a while, then sends a request to Server C (4). Server C receives the request (5), processes it, and sends return data (6). Server B receives the data (7), processes some more, then sends a request to Server D for another service (8). Server D receives it (9), processes it, and returns data (10). Server B receives the data (11), processes some more, and then returns data to client A (12). Client A receives the data (13), processes some more, then updates the user's display which completes the entire transaction (14). Of course Server B could have been implemented to call Servers C and D in parallel, instead of serially as shown in this example.

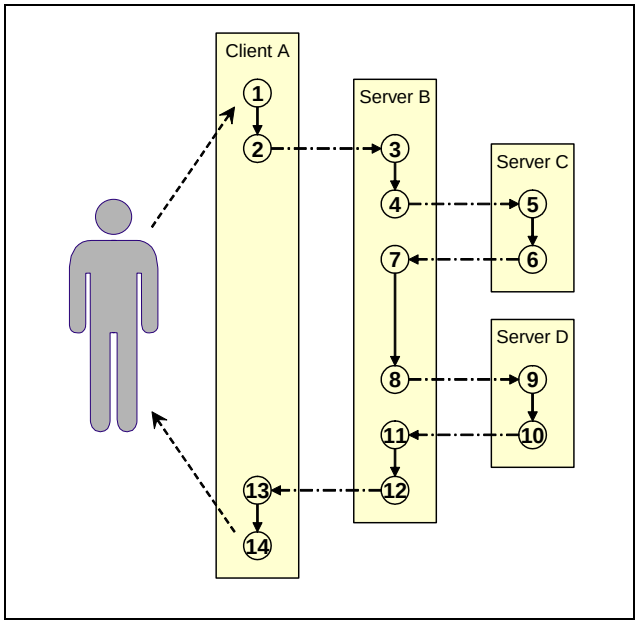

Figure 5 Nested Transaction Example

In this example, Client A takes the role of a client (though to the user Client A looks like a service provider), Server B takes the role of both a server (to Client A) and a client (to Servers C and D), and Servers C and D take the role of servers.

- The user thinks there was one transaction (1,14).
- Client A saw one transaction as a client (2,13).
- Server B saw one transaction as a server (3,12), and two as a client (4,7) and (8,11).
- Server C saw one transaction as a server (5,6).
- Server D saw one transaction as a server (9,10).

In order to understand how these seven transactions are related to each other, one needs to know the parent/child relationships.

- (1,14) is the parent of (2,13).
- (2,13) is the parent of (3,12).
- (3,12) is the parent of (4,7) and (8,11).
- (4,7) is the parent of (5,6).
- (8,11) is the parent of (9,10).

This list assumes that each point representing a transfer of control is measured. A less comprehensive set of measurements would not measure all the subtransactions, such as dropping (2,13), (4,7), and (8,11). One example of when this would be appropriate is if the elapsed time between 1 and 2 and between 13 and 14 are insignificant, so that measurements of (1,14) and (2,13) would be almost equal. This would yield the following parent/child relationships. (Other combinations are also possible, of course).

- (1,14) would be the parent of (3,12).
- (3,12) would be the parent of (5,6) and (9,10).

Learning relationship properties

This section describes the facility ARM uses to collect parent/child relationship information.

When indicating the start of a transaction with an arm_start_transaction(), the application can request that the ARM agent assign and return an identifier, named a correlator, for this instance of the transaction. A correlator is a value of typically about 100 bytes that is generated by the agent. The format is known to agents, and knowing the format is important for the analysis techniques described later. The application only needs to know the correlator length and need not be concerned with the format. The correlator may contain data to identify the transaction instance. It may contain other data that applies to all transactions in the call graph, such as business process and service class IDs.

The application is responsible for passing this correlator to server applications (that support this capability) along with the data needed to invoke the server transaction. The server application would pass any parent correlator on arm_start_transaction(). This allows the ARM agent to know the parent/child relationship.

Figure 6 shows the concept for a simple model. The principle can be extended to a model of arbitrary complexity. The arrows between the applications and

the ARM agents represent the arm_start_transaction() calls and the return of control to the application.

- Client A indicates the start of transaction T1 with an arm_start_transaction(), requesting a correlator. The agent returns correlator C1.
- Client A sends a request (reqB) to Server B, and includes C1 with the request.
- Server B starts transaction T2, indicates the same by calling arm_start_transaction(), passing C1 as the parent. At the same time it requests a correlator. The agent returns correlator C2.
- Server B sends a request (reqC) to Server C, and includes C2 with the request.
- Server C starts transaction T3, indicates the same by calling arm_start_transaction(), passing C2 as the parent.

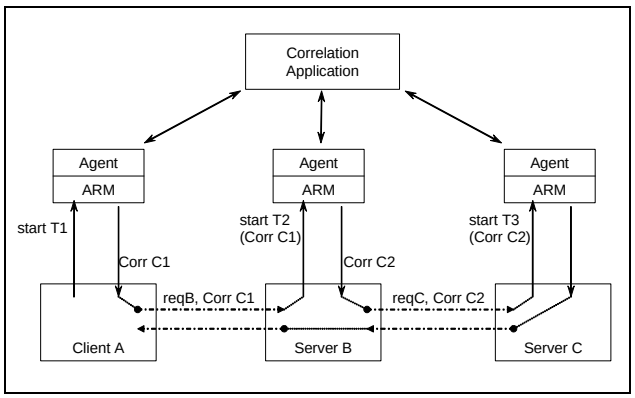

Figure 6 Use of Correlators

If the Correlation Application collects all the data about these transactions, it can put together the total picture, knowing that T1 is the parent of T2 (via C1), and T2 is the parent of T3 (via C2). This can be represented as T1→(C1)→T2→(C2)→T3.

An ARM agent may not provide correlators, either because it does not support the capability (it's optional), or because there is a management policy in effect is to suppress this information.

Analyzing end-to-end traces

The parent child relationships are often represented in a call graph, such as the one shown in Figure 7. In this example, four correlators (C1, C2, C3, C4) represent the similarly named transactions (T1, T2, T3, T4). Each correlator is passed to its child transactions, such as C4 being passed to T8 and T9, thereby indicating that T4 is the parent transaction of T8 and T9.

It is not unusual for transactions to contain many steps, and many of these may be instrumented with ARM. The resulting call graph can become rather large.

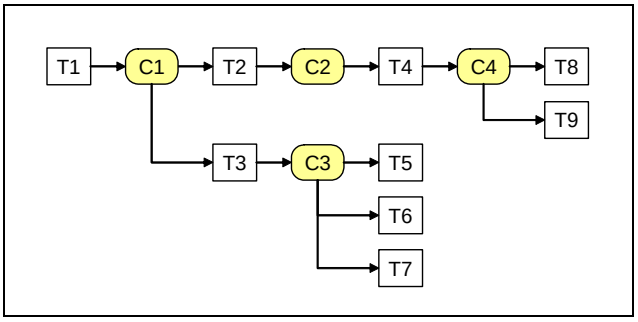

Figure 7 Example of an End-to-End Call Graph

Because there is a performance cost to performing an end-to-end trace, traces are generally only done on a sampling basis or when there is a known problem to address. For example, the monitoring software could be configured to run a full trace on 1% of all transactions. For these 1% correlators are generated and passed end-to-end to enable building the call graph. For the other 99%, the transactions may be measured for response time and status, and these data used for service level reports and threshold monitors, but no correlators will be generated or passed. The sampled traces can be used to establish a baseline that can then be used for periodic analysis.

Locating transaction failures

A common scenario is a failing transaction that is visible to an end user. The end user (or monitoring software) may inform the help desk, which triggers the analysis.

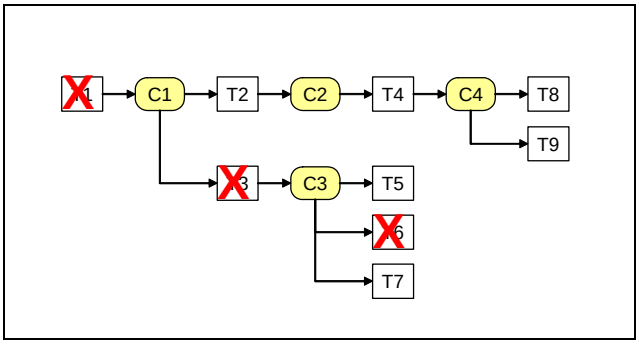

Figure 8 Locating a Failing Transaction

If a trace is available, or can be started for future transactions and the problem repeats, the specific

transaction (T6 in the example in Figure 8) that is causing the user's transaction to fail can be located.

Locating response time bottlenecks

Another common scenario, which may occur after an application has been deployed in production or during pre-production testing, is an unacceptably high response time. Again, the user experiences the problem and monitoring software may validate the problem, but this does not isolate where it is occurring.

By combining response time measurements with parent/child relationship information, an analyst can determine which transactions are executed the most often, and which are the biggest contributors to the response time seen by the end-user.

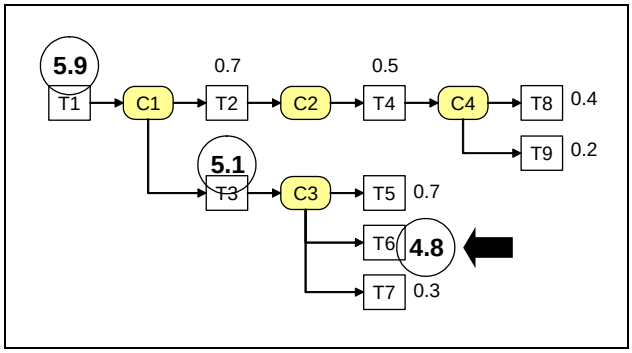

Figure 9 Response Time Analysis

The analyst may not find it cost-effective to analyze and tune every transaction for every application. The analyst can concentrate on those that will provide the most value if they are improved. By analyzing a call graph end-to-end, it is often easy to locate the specific transaction(s) that contribute the most to the response time. Further analysis and remediation can focus on improving the response time of those transactions. For example, Figure 9 shows a call graph in which T6 is the major contributor whose response time must be improved in order to improve the response time experienced by the end user.

Discovery of business system topologies

Another benefit to analyzing applications end-to-end is that it results in an understanding of how IT components are connected together in order to support a business process. As shown in Figure 10, ARM measurements result in understanding which components execute which transactions and how they depend on other components.

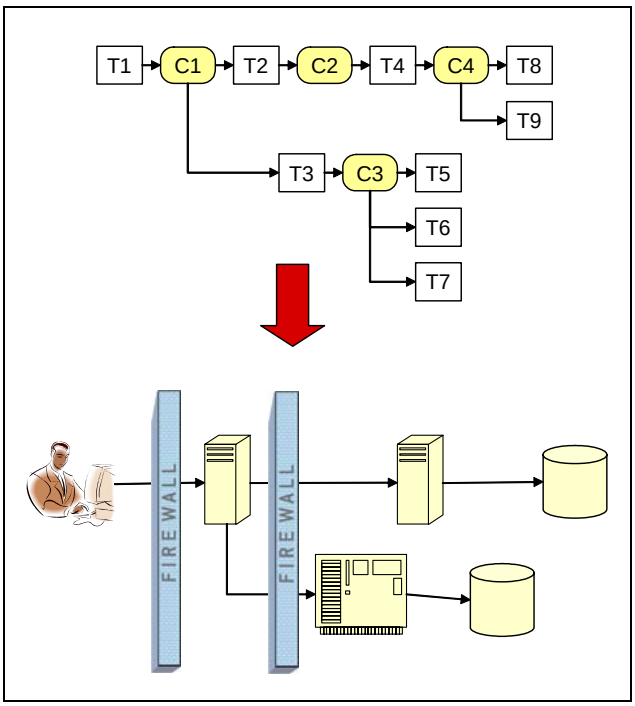

Figure 10 Discovering business system topologies

Topology information is useful for availability and performance management, but it also provides the base for many aspects of managing distributed applications. Aggregated over time, it aids in planning new deployments and asset accounting. As IT infrastructures become more dynamic, this type of auto-discovery becomes even more critical.

Analyzing ARM measurements

The techniques used to analyze the performance of distributed applications are similar to the techniques used to analyze single-system applications. The analyst is looking for bottlenecks, places where parts of the application are delayed waiting for slower part to complete, and ways to reconfigure the environment and/or the application to achieve faster response times, greater throughput, or both. Distributed applications are usually more complex. They run on multiple systems and are dependent on network connections between the systems. The systems themselves vary in configuration and processing power, sometimes dramatically so. The developers of a server program often don't know how that program will be used in a production environment.

The analysis process generally has multiple stages. Analysis starts at a macro level, focusing on the response time and throughput as seen by the end-user, and the response time and throughput of the major child transactions of the end-user transactions. The analyst is looking to understand the overall flow, and to reconfigure and tune the environment. In follow-on stages, the emphasis shifts to understanding performance within a given system, focusing on details like I/O rates, use of processes and threads, and so on. ARM measurements are useful in the macro level analysis, because they provide the perspective seen by the end-user and the major child transactions, and a way to relate the two to each other. For micro level analysis, kernel level measurements would be used heavily.

ARM itself does not create new analysis techniques. These techniques have been widely used for many years. The role of ARM is to enable these types of analyses by collecting critical data that has not been available for applications in distributed environments.

Example: Finding response time bottlenecks

A lot of useful analysis can be done by combining response time measurements with an understanding of the topology of the clients and servers, and the network connections between them. Consider the following scenario.

- 20 sites, each with 5 servers, and 25-100 clients,
- LANs (local area networks) connecting all the clients and servers within each site,
- WAN (wide area network) links connecting the different sites, which are much slower than the LAN links,
- a mix of two-tier and three-tier applications that consist mostly of shrink-wrapped commercial applications and a few that are developed in-house,
- applications that use ARM, and
- a workload which consists of a mix of transactions, some that can be served by a server at the same site as the client, and some that must be served by a remote server.

The analyst can influence several factors in order to optimize response times, throughput, availability, and cost.

- server locations – which applications run on which servers,
- server configuration, such as the amount of memory or processor speed,
- WAN link speeds,
- application tuning parameters, such as the number of threads that will be used and the priority of different transactions or users, and
- application design for the applications developed in-house.

The main technique available to the analyst is to summarize the data based on the factors that the analyst can influence, and then to correlate the response time data to the summarized data. In this scenario, the analyst would configure the ARM agents and associated management applications to summarize data by client site, by server site, by server application, and by server (the hardware system). Using analytical tools, sometimes even simple tools such as a spreadsheet, some very useful conclusions can be reached, such as the following examples. It is also useful to have historical data to compare against so the significance of degradations can be determined.

- If clients at one site are experiencing degraded response times, but clients at other sites are not, the problem is most likely local to that one site, or the WAN connection to other sites.

- If clients at one site are experiencing degraded response times for transactions to remote servers but not local servers, the WAN connection is suspect. If remote users of a server application at the local site are also experiencing degraded response times, this would corroborate the hypothesis.

- If response times for one application on a server are degraded, but response times for other applications on the same server are normal, the problem is probably with the one application. If workloads are normal so the problem isn't just congestion, it suggests a configuration problem. If this server is in a middle tier and is dependent on other servers, these other servers should also be checked to see if they are the source of the unusual delay.

- If response times for all the applications on a server are degraded, then the most likely causes are congestion due to heavy workloads, a hardware problem, or a system configuration problem. Other possibilities, if all these applications are in a middle tier and are dependent on other servers, are the other servers, or the network connections to these servers if they are remote.

- If response times for all the applications and all the servers on a site are degraded, the most likely cause is a LAN problem or congestion. If the slow response times only affect remote users, the most likely cause is a WAN problem or congestion.

- If there is a server application that runs on multiple servers in multiple sites, and response times are degraded for all users regardless of where they are located, this suggests either an application configuration problem, or a secondary problem such as a problem with a database server used by all these server applications.

Tuning application configuration

Applications are written once but must function effectively in many different environments. There will be demands for system resources from many applications. One approach to providing the needed flexibility is to provide ways to tune the configuration, such as ways to control the number of buffers allocated and the number of threads used. Response time data can be collected and correlated with these parameters. By looking for elbows in the curves, an optimal value can be found.

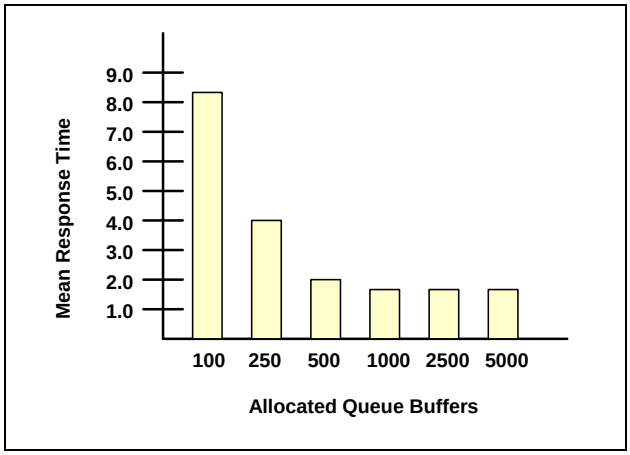

Figure 11 Analyzing Response Time vs. Environmental Factors

In other cases, applications don't provide tuning controls. The performance would still be affected by system configuration parameters. The amount of memory on a system is a common parameter that often has a dramatic affect on observed response times. By correlating response time data with these parameters and looking for elbows in the curves, an optimal value can be found.

Applications can use ARM to provide information useful for tuning. For controllable parameters such as the number of buffers or threads used, or non-controllable parameters such as the number of connected users or the length of internal queues that represent congestion, the application can provide measurements of the actual values as detailed properties. By correlating these measurements with the response time measurements, and looking for elbows in the curves, optimal values can be found. Figure 11 is an example analyzing response time vs. queue buffers.

Organizing Data

When doing end-to-end correlation, it is essential that the solutions be scaleable. An example of a solution that would not be scaleable is saving a record for each and every transaction instance in a trace mode, then sending all the records to a common analysis point, and then doing the analysis. This is practical when traces are collected on an exception basis. Sampling is a useful way to control the amount of data that is stored. Another is to summarize the data as it is received and save only the summaries. This granularity is often sufficiently fine for many purposes.

Well designed correlators allow ARM agents to summarize data across many transaction instances rather than saving data about each instance. This is done by providing within the correlator classification data, such as business process ID and service class.

In a typical usage scenario, an ARM agent would summarize data at an application server for one or more of the following: all instances of a transaction, all instances from each network address and/or each user, and all instances from each unique parent transaction (this server transaction could be invoked by many different parent transactions and parent applications). The number of data records to be processed would be a manageable amount under most conditions.

An ARM application that manages many ARM agents would provide additional summarization by site, by application, and by application server. This is the type of summarization described in the "Finding response time bottlenecks" section.

Summary

In the network computing world we are now in, managing applications is a key challenge to solve.

Comprehensive solutions are needed that include administrative tasks, monitoring at the application level, and monitoring the transactions of individual users. The ARM standard is well positioned to play a key role collecting the information needed to manage service level agreements and to analyze the performance of distributed applications. Far-sighted developers who make the investment necessary to "ARM" their applications will give their customers far greater insight and control over their application environments. More and more development and management tool vendors are delivering solutions based on ARM, insuring that the users of these applications will have a wide choice of solutions available.

ARM information can be combined with other sources of similar information for both of these purposes. This paper has described ARM, and shown how an application can use ARM. From the management perspective, this paper has shown how ARM information would be used, and how the management solutions that need to be developed are scaleable and not overly complex.

References

[ARM98] ARM 2.0 Technical Standard, available from www.opengroup.org/tech/management/arm/

[ARM01] ARM 3.0 Java Binding Technical Standard, available from www.opengroup.org/tech/management/arm/

[ARMJ03] ARM 4.0 Java Language Binding Technical Standard, available from www.opengroup.org/tech/management/arm/

[ARMC03] ARM 4.0 C Language Binding Technical Standard, available from www.opengroup.org/tech/management/arm/.

[EWLM04] Enterprise Workload Manager, available from http://publib.boulder.ibm.com/eserver/v1r1/en_US/index.htm?info/ewlminfo/kickoff.htm

Trademarks