The Application Response Measurement (ARM) API, Version 2

Mark W. Johnson
Tivoli Systems

The ARM API provides a way to manage business transactions. By embedding simple calls to an agent supporting the ARM API, an application can be managed for availability, service level agreements, and capacity planning. Version 1 of the ARM API and a software developer's kit was made available in June 1996. A group of end users, tool vendors, and business application vendors have defined Version 2 of the API. This paper describes versions 1 and 2, how agents are implemented, and how applications can best exploit the API.

Introduction

During their wide-spread introduction in the 1960s and 1970s, computer systems had a dramatic impact on many businesses. The widespread 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. In the 1990s, the greatest advantages are being realized by organizations that exploit network computing. Network computing combines PCs and workstations, powerful shared computers, large databases, and networks that connect them to provide a platform for new applications that were unthinkable even a few years ago.

These applications provide unprecedented opportunities for organizations to reach more customers with ever more useful services. These services are critical for success in a market. The applications boost productivity dramatically 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 need to do? Are transactions completing successfully? Are response times satisfactory? If not, where are the bottlenecks?

The Application Response Measurement (ARM) Application Programming Interface (API) is a way for applications to answer these questions. The applications can provide critical information about business transactions from the perspective of the business operations (rather than, say, the network). 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.

What is the ARM API?

The ARM API is a simple API that applications can use to pass vital information about a transaction to an agent. Simplifying slightly, all the application has to do is call the ARM API just before a transaction (or a subtransaction) starts and then again just after it ends. Figure 1 shows how business applications are managed with the ARM API. The API is supported by an agent, which measures and monitors the transactions, and makes the information available to management applications.

The ARM calls identify the application, the transaction, and (optionally) the user, and provide the status of each transaction when it completes. This is sufficient information for the management solution to answer vital questions. 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 instantaneous notification when a transaction is not completing satisfactorily, and appropriate steps taken to remedy the situation. Alternatively, this information can be logged and accurate counts tracked over time.

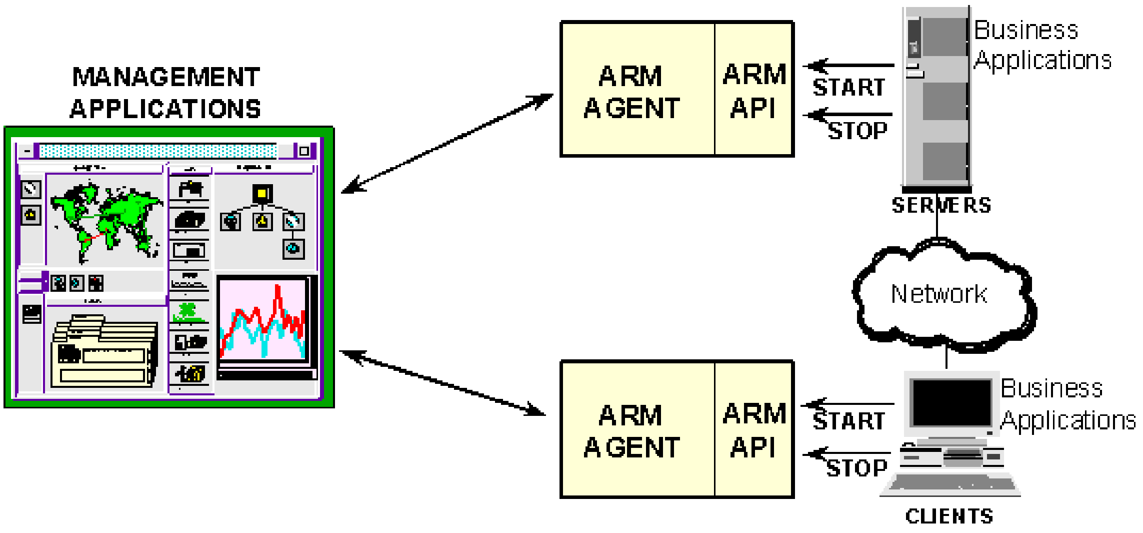


Figure 1. Overview of the ARM API

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, these are the only two things that really matter: is it working, and how long is it taking? Response times are important elements in service level agreements, a capability that was almost taken for granted with host-centric applications, but which has been rarely implemented with distributed applications, because of the difficulties of doing so.

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 is 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 subtransactions 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 subtransactions 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.

History and Deliverables

What was announced as the ARM API in June 1996 started as separate and independent projects at Tivoli Systems (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. 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 the Application Response Measurement API. As the work progressed towards announcement, many users and vendors were consulted, so that by the time the ARM API was announced in June 1996, there was a long list of companies supporting the activity.

Today the ARM Working Group is comprised of BGS, BMC, The Boeing Company, Boole & Babbage, Candle, Citicorp, Compuware, Hewlett Packard, IBM, Landmark, Novell, Oracle, SAS, SES, Sun, Tivoli Systems, Unify, and Wells Fargo. The ARM Working Group works with standards organizations to ensure that the ARM API will continue to be a valuable standard for monitoring application performance and measuring IT service levels.

The ARM Working Group has produced a Software Developers Kit with three important deliverables. The developers toolkit is available on CD-ROM or it can be downloaded from web sites (http://www.cmg.org, http://www.tivoli.com or http://www.hp.com).

1. The specification of the API is described in an API Guide, which is available in both softcopy and hardcopy. All of the six calls and their respective parameters are described in detail and documented in a style familiar to C programmers. Samples are provided. The API can be called from many languages, such as Smalltalk and Visual Basic.

2. Stub libraries are provided that an application can link to and call. The stub libraries don't do anything with the information in the calls - they just return immediately with a return code of zero. Their primary role is to be shipped with an application that makes ARM calls so the application can run even if there isn't an ARM agent installed. To monitor an application using ARM, the stub libraries would be replaced with an agent that supports the API, and which actually uses the data to monitor the applications. The stub libraries are available for Windows 3.1, Windows 95, Windows NT, OS/2, IBM AIX, Sun Solaris, HP-UX, and NCR MP-RAS.

3. Included in Version 2 of the SDK is a simple agent that will log all ARM calls to a trace file and that can be used for testing the instrumentation in an application. It is provided in source code format, so it can be compiled (and modified if necessary) for any platform.

Design Criteria

Because the ARM API is only useful if applications actually call it, certain design criteria were rigorously adhered to during the design of the API.

The API is simple to use. The number of calls are few, and few parameters are required. It can be called from many different programming languages, and application development tools can put the calls in applications they generate.

The runtime overhead is low, and is insignificant compared with the great value of using the API.

The API is designed to be extendible while still being fully backwards compatible. This means that anyone who instruments their application to call the ARM API can be reassured that rework of their application will not be needed just to maintain the existing management capability, even if the API is enhanced. Exploiting the newest features will be optional. Any implementation using Version 1 of the API is fully supported by Version 2 implementations.

How to Use the API

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

1. The first step is to define the key business transactions. This is the most important step. Each application developer needs to define 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.

2. The second step is to modify the program to include calls to the ARM API. The stub libraries and logging agent in the developers toolkit can be used for initial testing. Because the API calls are simple, this step is neither difficult nor time-consuming. The key is to decide where to put the probes in the first place, by doing a good job defining the key business transactions.

3. The third step is to replace the stub libraries from the developers toolkit with an ARM-compliant agent and associated management applications. The distributed applications will now be monitored in ways that could previously only be hoped for.

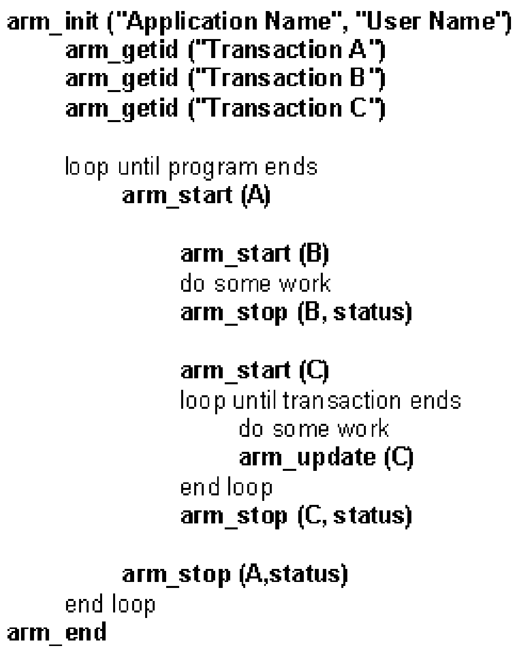
There are two kinds of transactions that will provide the greatest benefit if they are instrumented. The following procedure is suggested.

Start with transactions that are visible to users or that represent major business operations. These are the building blocks for service level agreements, for workload monitoring, and for early problem detection.

Next focus on transactions that are dependent on external services, such as a database operation, a Remote Procedure Call (RPC), or a remote queue operation. These will generally be subtransactions 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.

Version 1 of the ARM API

This section contains a brief description of Version 1 of the API. A complete description is included in the developers toolkit. There are six API calls. The workhorses are arm_start and arm_stop, which are called at the beginning and end of each executing transaction. There are two administrative calls used to define the application and transactions: arm_init and arm_getid. arm_update is an optional call that can be used with long running transactions to provide an "I'm alive" heartbeat and a progress indicator. arm_end is used when an application is shutting down, which allows an agent to release any storage used for monitoring this application. Figure 2 is an example in an imaginary language showing the API calls. Working examples can be found in the ARM API Guide. Following is a more complete description of each of the calls.



```
arm_init ("Application Name", "User Name")
    arm_getid ("Transaction A")
    arm_getid ("Transaction B")
    arm_getid ("Transaction C")

    loop until program ends
        arm_start (A)

            arm_start (B)
            do some work
            arm_stop (B, status)

            arm_start (C)
            loop until transaction ends
                do some work
                arm_update (C)
            end loop
            arm_stop (C, status)

        arm_stop (A,status)
    end loop
arm_end
```

Figure 2 - Conceptual Example of ARM API Calls

arm_init defines the application name and (optionally) the user ID. It is typically executed once while the application initializes. The application name and user ID are specified as character strings. The main reason for defining an application is to help organize the information, and to minimize the problem of having duplicate transaction names defined in different applications. The return code from arm_init is a unique identifier generated by the agent; it is passed as a parameter on arm_getid and arm_end calls.

arm_getid defines the transaction name and (optionally) details about each transaction. arm_getid is called once for each transaction name and is typically executed while an application is initializing. There can be any number of transaction names defined for each application. The transaction name and details are specified as character strings. A transaction name needs to be unique within an application. The combination of application name and transaction name uniquely identify a transaction class. The identifier returned on the arm_init call is passed as a parameter on arm_getid so the transaction name will be associated with the correct application. The return code from arm_getid is a unique identifier generated by the agent. It is passed as a parameter on arm_start calls. This identifier is unique for all transaction classes within a system.

arm_start indicates that an instance of a transaction has begun execution. Contrast this with arm_getid, which defines the name of the transaction class during initialization, but doesn't actually indicate that a transaction is executing. The identifier returned on the arm_getid call is passed as a parameter on arm_start so an agent knows which type of transaction is starting. There can be any number of instances of the same transaction class executing simultaneously on the same system. To identify each one, the return code from arm_start is a unique handle generated by the agent. This handle is unique within a system across all instances of all transactions of all applications. The transaction instance is the fundamental entity that the ARM API measures.

arm_update is an optional call to indicate that an instance of a transaction is progressing, or at least is still active (a heartbeat). It would typically be used for a transaction that takes many minutes or hours to complete. An update might be sent at a fixed time interval (e.g., every minute) or after a fixed amount of work (e.g., after every 1000 records are processed). The identifier returned on the arm_start call is passed as a parameter to indicate which transaction instance is being updated.

arm_stop indicates that an instance of a transaction has completed. The handle returned on the arm_start call is passed as a parameter to indicate which instance of a transaction is ending. arm_stop can be issued from a different thread or process from which the arm_start was issued. The transaction status is also passed, and there are three possible values.

> **GOOD -** the transaction completed normally and the intended service was performed.
>
> **ERROR -** the transaction completed with an error (so the intended service was not performed).
>
> **ABORT -** the transaction was unable to complete. An example of a reason for an abort would be a time-out error from a communications stack.

arm_end indicates that the application will not be making any more calls to the ARM API. It is typically used when an application is shutting down. Any information about transactions that are in-process (an arm_start has been issued but not yet a corresponding arm_stop) would be discarded.

Performance Considerations

There are some practical considerations that affect the design of agents and applications implementing the ARM API. Consider the internal structure of a typical ARM agent.

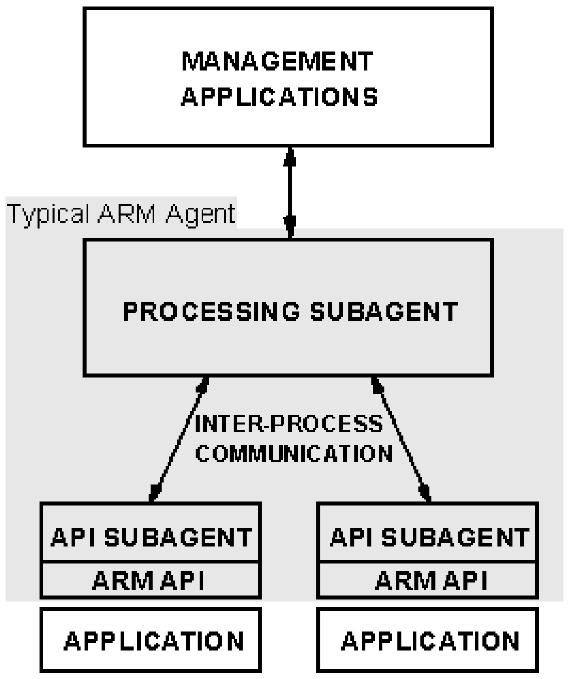


Figure 3 - Anatomy of a Typical ARM Agent

The application is dynamically linked to part of the agent, labeled API Subagent in Figure 3. The API Subagent runs in the process (address) space of the application, so it is kept as small and fast as possible. There is one instance of an API Subagent for each instance of an application linked to the ARM library. The API Subagent's job is to capture the ARM calls from the application and put them on an inter-process communications (IPC) channel to the Processing Subagent, which does most of the work. The API Subagent also returns to the application handles and status codes generated by the Processing Subagent.

The Processing Subagent runs as a separate process (or more than one process) with its own address space. It manages all the handles, processes all the data, and interfaces to management applications. There is one Processing Subagent on a system. Actually, there's no reason more than one Processing Subagent couldn't be installed on a system, but each API Subagent works with only a matching Processing Subagent.

There are two main considerations for achieving optimal performance: minimizing application delays and minimizing system overhead. Of the six ARM calls, arm_start, arm_update, and arm_stop are the ones that really matter from a performance perspective. arm_init, arm_getid, and arm_end are used generally only when a program initializes and shuts down, so performance isn't critical.

Minimize Application Delays. Every time the application calls the API Subagent, it is delayed from its processing. If the application is single-threaded, the calling sequence will be something like what is shown in Figure 2. The application is unable to execute the subtransaction while waiting for the arm_start and arm_stop calls to complete processing. Minimizing the path length is of paramount performance. In particular, an agent design that avoids waiting for any IPC operations during processing of arm_start, arm_update, and arm_stop is highly desirable. The API Subagent should simply post the information for the IPC operation (to execute later) and return to the application.

Applications are encouraged to make ARM calls inline, rather than making the calls to the API Subagent from a separate thread. Although it seems like this would be a way to avoid IPC delays, regardless of the design of the agent, there are significant disadvantages with this approach.

- It increases system overhead.
- It increases program complexity because the application has to manage the extra thread.
- It creates variability in the measurements because one won't know whether the application's

transaction executed before or after the arm_start and arm_stop calls, or by how much.

Because of these disadvantages, this approach is strongly discouraged. It is much better to have a good and fast ARM agent and make the calls inline.

Minimize System Overhead. Whenever an agent is executing, obviously the CPU can't be executing application transactions. This isn't likely to be an issue on a desktop client, which will have many idle cycles waiting for transactions sent to servers to complete, and for users to think. It could be critical on a heavily loaded server, because there aren't many idle cycles. The best agent for this environment is one that is optimized to minimize total path length. The agent should also have an even shorter path length if the data is not being monitored or processed. This provides an administrator with a way to control total overhead, by reducing the number of transactions monitored or the amount of data processed and logged.

Changes in Version 2 of the ARM API

The initial version of the ARM API is robust and meets a large set of requirements for measuring application performance. These measurements can be used for service level agreements, threshold monitoring, availability monitoring, and to measure workloads for capacity planning and charge-back accounting. These measurements tell a great deal about WHAT is happening, but not so much about WHY it is happening. Version 2 extends the ARM API to help answer WHY things are happening, such as answering the following questions.

- Where are delays occurring? Which transactions are holding up other transactions?
- How "big" is a transaction, such as what is the size of a file being transferred? Is the application congested? How is an application using resources?
- When something goes wrong, what diagnostic information can point to the problem?

All Version 1 API calls have three reserved parameters: a pointer to a data buffer, the length of this buffer, and a word of flags. Version 2 uses the buffer pointer and buffer length parameters to pass the following additional information. See the ARM API Guide for details about how to use the parameters.

- The parent/child relationship between transactions is represented by correlators passed on an arm_start call.
- Application and transaction metrics, and diagnostic information can be passed on arm_start, arm_update, and arm_stop calls.

If these two parameters are set to zero, which they must be in Version 1, the API functions identically to Version 1. This means that the extensions are completely backwards compatible, and applications using Version 1 of the API do not need to be recompiled to work with an agent supporting Version 2 of the API.

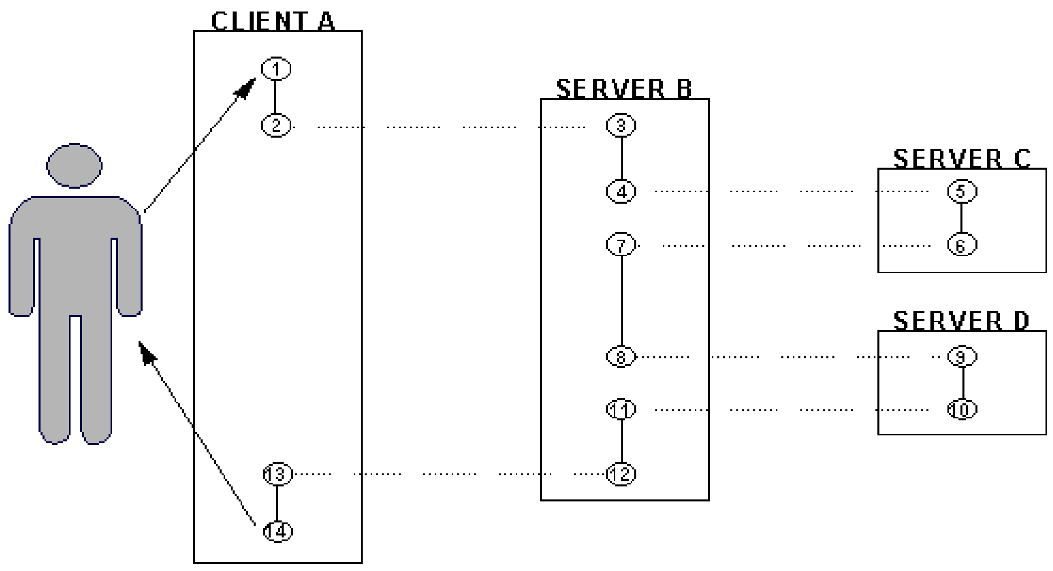


Figure 4 - Nested Client/Server Transactions

Using ARM to Correlate Transactions and Subtransactions

Version 2 provides a way to correlate transactions using a client/server programming model. 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 4 is an example.

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, 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.

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 C saw one transaction as a server (5,6).

Version 1 of ARM provides a way to measure each of these seven transactions, but not any way to understand how they are related to each other. To understand this, one needs to know all 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 from 1-2 and 13-14 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).

How are correlators collected with ARM? Where are delays occurring? The semantics of Version 1 measure transactions without regard to whether they are composed of other transactions. In practice, many client/server transactions consist of nested subtransactions. It's very useful to know that a transaction is slow, but even more useful to know which subtransaction(s) contribute most to the delays. Using Version 2 an application can provide the parent/child information needed to know how transactions and subtransactions relate to each other. There are two facilities to collect correlation information.

1. When indicating the start of a transaction with an arm_start, the application can request that the ARM agent assign and return a correlator for this instance of the transaction. The ARM agent has the option of not providing the correlator, either because it does not support the capability (such as if it's a Version 1 agent), or because the management policy in effect is to suppress this information.

2. On the same arm_start, the application can provide a correlator for a parent transaction. This allows the ARM agent to know the parent/child relationship.

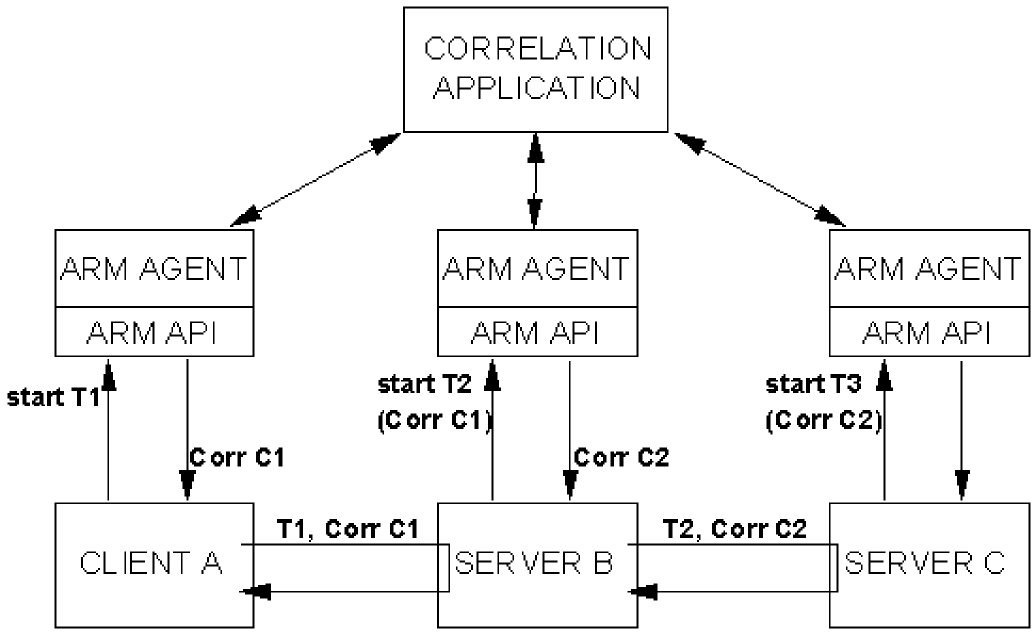


Figure 5 - How Transactions are Correlated using the ARM API

Figure 5 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 calls.

- Client A starts transaction T1, requesting a correlator, and is assigned C1.
- Client A sends a request (T1) to Server B, and includes C1 in the request.
- Server B starts transaction T2, passing C1 as the parent. At the same time it requests a correlator and is assigned C2.
- Server B sends a request (T2) to Server C, and includes C2 in the request.

- Server C starts transaction T3, passing C2 as the parent.

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.

In order to enable scaleable solutions, ARM agents follow conventions when creating correlators. The format is flexible and extendible so more conventions can be added if the need arises. Correlators contain the following information:

- Flags (one flag is defined - a "trace this correlator" flag)
- Transaction class ID (transaction id returned from an arm_getid)
- Transaction instance (start_handle returned from an arm_start)
- Agent Location/ID. This field identifies the system the agent is running on, as well as addressing information. Examples are a network address plus a port number.

How would correlation be used in practice? A concern when tracing any application flows is overwhelming a correlation application and/or the network with the volume of data collected at the agent, sent to the correlation application, and processed. Management applications need to use tracing selectively as part of an overall strategy. The format of the correlator can be exploited to filter the amount of data to trace and process, as shown in the following scenario.

Monitoring the application starts with monitoring the transactions, like can be done with Version 1 of the ARM API. Data would be collected on clients and compared to thresholds. The data might be logged for trending analysis as well. As long as the response times and availability were satisfactory, no correlation analysis would be done.

If response times at a client began to be unacceptable, the Correlation Application (which is also monitoring service levels) would instruct the agent at the slow performing client to turn on the trace flag for all transactions. This client is represented in Figure 5 by Client A. As the correlators are passed from system to system, each ARM agent sees the trace flag so it knows to capture the correlation information, and it knows to turn the trace flag on in the correlators it generates. These servers are represented in Figure 5 by Server B and Server C. Correlators sent from other systems don't have the trace flag on so they are not captured nor are the trace flags turned on in their child transactions. This results in data being traced and processed for transactions started at clients with poor performance, but not for any other transactions.

Because only the transactions of interest are being traced, it is practical for the agents to forward the trace information to the correlation application as it is collected or at frequent intervals. The correlation application connects the transactions together using the parent/child relationships and looks for trouble spots. If samples of trace data have been collected over the past couple of months and analyzed, a baseline of expected performance at the subtransaction level would be available, which would make the search for trouble spots easier.

If the ARM agents on the slow systems generate correlators routinely, but don't make available an online control for the trace flag, the agents can exploit the fact that the agent (system) ID is included in the correlator. The correlation application could provide the ARM agent at servers with a list of client systems that are experiencing slow response times. The servers might be selected by the correlation application after consulting an inventory database showing application connectivity. The agents at the servers use the Agent ID field in the correlator to compare with the list of systems, and capture all the correlation information for transactions from those systems. It does not capture correlation data from other systems. It could also turn on the trace flag for correlators it generates, when the parent transaction is from one of the systems being traced.

Using ARM to Collect Application and Transaction Metrics

If performance and availability are not acceptable it's useful to look inside the application to determine why. Of course this could be done with a debugger, but in an application that is already in production use, this is not a practical approach. An alternative that is practical is to have the application provide a few key pieces of information. Version 2 provides a way for applications to do this, such as the following examples.

How "big" is a transaction? Measuring a transaction such as a file transfer is problematic because the response time varies greatly depending on the size of the file being transferred. With Version 2, an application can provide additional information about a transaction such as a count of bytes or records processed. This information could be used to specify thresholds not as an absolute number (such as 30 seconds), but rather as a ratio such as one second per kilobyte. Another use would be to create a histogram of the number of jobs of differing sizes, as shown in Figure 6.

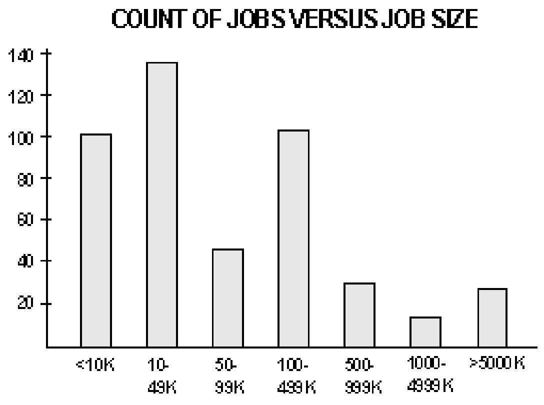


Figure 6 - Example of Using Transactions Metrics

This could be done in a more cumbersome way with Version 1 by defining different transactions for jobs of different sizes, such as FileTransfer_10K, FileTransfer_49K, FileTransfer99K, etc. The drawback is a loss of granularity, because the same threshold would have to be used for jobs of 100KB and 499KB, for example. This also requires that the application know the size of the transaction before it starts processing, so the correct transaction class is selected.

How can applications and configurations be tuned to improve performance? The application can provide information about the level of resources available to process the transaction, such as the number of buffers allocated, or the number of threads or queues being used. By correlating response times with the resources allocated, the administrator can get a much better idea of how to configure applications and their environment. The application can provide information about internal congestion within the application, such as the number of transactions in-process or waiting to be processed. This information can be monitored with thresholds and a notification sent to an operator or an automation routine when congestion is detected. This information could also be tracked over time and correlated with the observed response times to get a good idea about whether congestion is having significant affect on response times. If it is, workloads or configurations can be adjusted to improve performance.

How are application and transaction data passed across the ARM API?

Applications can pass on arm_start, arm_update, and arm_stop six fields of 8 bytes each, and one string of 32 characters. This information is in addition to the correlation information. The format of the buffer is shown in Table 1.

Header	8 bytes
Metrics and Strings (six 8 byte fields)	48 bytes
Character String (one 32 byte field)	32 bytes
Correlator (current formats)	26-40 bytes

Table 1 - Format of Buffer passed in arm_update call.

The metric fields are 8 bytes long and can contain:

- one 32 bit integer
- one 64 bit integer
- two 32 bit integers that form a ratio
- one string of 8 characters

The integer fields can represent:

- a counter, such as bytes transmitted or records processed.
- a guage, such as queue length or memory allocated
- a number not intended for arithmetic operations, such as a message number

The 8 byte character strings would typically be used for things like a sense code or an LU name. The 32 byte character string might be used for a part number, account code, or an error message.

The format, name, and position of each field are defined on the arm_getid call, and cannot be changed. Each field can be passed on any or all of the arm_start, arm_update, and arm_stop calls. Flags on each call specify whether the field is included.

Using ARM to Collect Diagnostic Information

When a problem or unusual event occurs with a transaction, the application often has useful knowledge. Examples would be an error code from a server or the account number being processed by a failing transaction. The problem or event may cause the transaction to fail or to be delayed substantially. The application can always log this information, but this makes it difficult to relate the diagnostic information with the transaction that was executing when the problem occurred. Version 2 defines a way for an application to provide diagnostic information linked to transaction measurement data.

There are two ways to provide diagnostic information. One is to use the 8 byte and 32 byte character strings that can be passed on the arm_start, arm_stop, and arm_update calls (described in the previous section). Often diagnostic information like sense codes or account numbers can be put in one of these fields and this satisfies the requirement.

The amount of information that can be passed on the arm_start, arm_update, and arm_stop calls is intentionally limited so agents can process these calls as efficiently as possible. If applications need to pass more data, a special form of arm_update can be used. Up to 1024 bytes can be passed, with the first four bytes used to specify the format, and the rest of the data determined by the application.

Application developers are advised to use the 1024 byte arm_update format with restraint. The agent does not understand the semantics of the data so all it can do is trace the data. Being able to provide this service to an application and to be able to relate this information to the ARM transactions is a useful function, but the system overhead needs to be considered. The data must first be passed using the IPC, then written to a trace file, sent across the network, or processed in some other way. Using this call on a busy server should be discouraged. Agents can also permit administrators to turn on and off the processing of this data independently of normal ARM processing. Because the data format is free-form, agents can't do much with the data except log it. A suggestion is to use human-readable character strings so the data can be easily understood.

Summary

In the network computing world of the late 1990s, 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 API is well positioned to play a key role in monitoring. Version 1 of the API met many requirements for measuring and monitoring transactions. Version 2 extends this capability by enabling powerful solutions for diagnosing the performance of applications while they are running in production. Far-sighted developers who make the small investment necessary to ARM their applications will give their users 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.

Updated December 1997.