ARM (Application Response Measurement) is an emerging standard for measuring the performance and availability of business transactions and their components. A third version of the standard, ARM 3.0, is expected to be available in the first half of 2000. Enhancements include flexible transaction registration, a loosely coupled interface for simpler programming, and native Java bindings. This paper describes the new enhancements and proposes some other enhancements that may be in a version 4.0 of the standard. The relationship of ARM to standards of The Open Group and the DMTF (Distributed Management Task Force) are also discussed.

1 Introduction

It is hard to imagine conducting business around the globe without computer systems, networks, and software. People distribute and search for information, communicate with each other, and transact business. Computers are increasingly faster, smaller, and less expensive. Networks are increasingly faster, have more capacity, and more reliable. Software has evolved to better exploit the technological advances and to meet demanding new requirements. The IT infrastructure has become more complex. We have become more dependent on the business applications built on this infrastructure because they offer more services and improved productivity.

No matter how much applications change, administrators and analysts responsible for the applications care about the same things they have always cared about.

- Are transactions succeeding?
- If a transaction fails, what is the cause of the failure?
- What is the response time experienced by the end user?
- Which sub-transactions of the user transaction take too long?
- Where are the bottlenecks?
- How many of which transactions are being used?
- How can the application and environment be tuned to be more robust and perform better?

ARM helps answer these questions. ARM is an emerging standard for measuring service levels of distributed applications (see [ARM97] and [JOHN97]). ARM measures the availability and performance of transactions (any units of work), both those visible to the users of the business application and those visible only within the IT infrastructure, such as client/server requests to a data server.

1.1 The Evolution of ARM

ARM 1.0 was developed and released in June 1996. ARM 2.0 was developed in 1997 and approved as a standard of The Open Group (see [TOG98]) in July 1998, part of the IT DialTone™ initiative. The technical specification for a third version of the standard, ARM 3.0, is essentially complete. The target date to release a final specification and a software developer’s kit for ARM 3.0 is early 2000.

1.2 Highlights of ARM 3.0

This paper describes ARM 3.0. ARM 3.0 contains several enhancements that make ARM easier to deploy and maintain, and that provide more capabilities.

- Transactions are defined using static identifiers that do not change from system to system.
- A loosely-coupled interface simplifies programming in some environments and offers the option of making one call each time a transaction executes instead of two calls.
- Native Java bindings extend ARM to support the popular programming environment. Native bindings
avoid the need to use the JNI (Java Native Interface) to call ARM from Java programs.

ARM 2.0 and ARM 3.0 provide equivalent function. The difference is in the mechanics of how that function is provided. If ARM 2.0 meets the application requirements, there is no reason to use ARM 3.0. To support the ARM 3.0 mechanics the transaction semantics are subtly different. There is a new format for transaction identifiers and transaction handles, and new API calls.

2 Overview of ARM 1.0 and 2.0

ARM 1.0 and ARM 2.0 semantics are identical except ARM 2.0 supports optional attributes and unit of work tokens (known as “correlators” in the ARM specification) to drill down from parent to child units of work (transactions). Because ARM 1.0 is a pure subset of ARM 2.0, the remainder of the paper refers to ARM 2.0.

Applications using ARM define transactions (any unit of work) that are meaningful within the application. Typical examples are transactions initiated by a user and transactions with servers. Applications on clients and/or servers call the API when transactions start and stop. The application provides a completion status on the stop call. A measurement agent implementing the ARM API measures the elapsed time between the arm_start() and arm_stop() calls. It is essential that the arm_start() and arm_stop() calls are made synchronously inline as close as possible to the time the transaction actually starts and stops. It is an egregious error for an application to make arm_start() and arm_stop() calls from a different thread being independently scheduled because the response time measured by the agent will almost certainly be incorrect.

Figure 1. ARM 2.0 Overview

Management applications collect information from ARM agents on different systems to put together an end-to-end picture of a transaction. The end-to-end picture starts with the transaction visible to the end-user on the client system. The client application indicates the start of a transaction to the ARM agent and requests a “correlator”, which is a token identifying the unit of work. The token is an opaque data field about 30 bytes long. The client application passes the correlator to the server application. The server application passes the client's correlator (which indicates its parent) to the ARM agent when it starts its transaction. The process can repeat any number of times within and across systems.

The API has six calls. The syntax is familiar to C programmers.

- **arm_init** Registers the application name and optionally the user name.
- **arm_getid** Registers the static attributes of a transaction, including its name and any optional attributes.
- **arm_start** Indicates that an instance of the transaction has begun executing.
- **arm_update** A heartbeat that indicates a transaction instance is still processing. This call is optional.
- **arm_stop** Indicates that a transaction instance has completed.
- **arm_end** Indicates that no more ARM calls will be made by this application.

2.1 Measuring Transaction Instances

ARM measures transaction instances. Response time and status are measurements that apply to every instance. The optional attributes may be measurements also. In other cases the optional attributes are information such as a string or error number.

2.1.1 Response Time

The agent calculates the response time. It is the elapsed time between the arm_start() and arm_stop() calls. It is essential that the arm_start() and arm_stop() calls are made synchronously inline as close as possible to the time the transaction actually starts and stops. It is an egregious error for an application to make arm_start() and arm_stop() calls from a different thread being independently scheduled because the response time measured by the agent will almost certainly be incorrect.
2.1.2 Status

The application provides the status. The three possible values are Good, Failed, and Aborted. An example of aborting a transaction is pressing the BACK or STOP button in a browser or selecting a link before a page fully loads.

The seven optional attributes can be provided on any and all of arm_start(), arm_update(), and arm_stop(). The definition (name and format) of each attribute is provided as a static attribute on arm_getid().

3 Defining Transactions

Transactions exist in two contexts. The first is the definition of the transaction’s static attributes. The second is each instance of a transaction when it executes. This section describes transaction definitions and the next section transaction instances.

3.1 Defining Transactions in ARM 2.0

Before a transaction can be measured its static attributes are defined. There are two required static attributes and several optional static attributes.

The two required attributes are the name of the application, provided on arm_init(), and the name of the transaction, provided on arm_getid(). Figure 2 shows the calls. Together they can be thought of as the key of the transaction definition. Each name is a character string up to 128 bytes long. The combination of the two values must be unique to avoid name collisions. An example of an application name is “MegaBank Banking Application 2.3”. An example of a transaction name is “Query Account Balance”.

The names are mapped to a transaction identifier, a 32-bit integer dynamically assigned by the agent as the return code of the arm_getid(). The concept is similar to a file handle returned when an application opens a file by name, but afterwards refers to the file by the integer handle. One characteristic of this method is that the identifiers assigned by the agent will usually vary from system to system, even though the application and transaction names are the same.

One of the optional static attributes is a user identifier, provided on arm_init(). The term “user identifier” is ambiguous – is it the person’s name (John Smith), the logon name (jsmith), or a numeric index assigned by the operating system or application? In actual practice this attribute is often not used or is used for completely different purposes. An example is identifying the target server in a program that proactively tests a server by executing typical transactions.

The other optional static attributes are the name and data type of attributes that can be provided about each transaction instance when it executes. For example, the name of an attribute might be “Number of Records Processed” and the data type a 32-bit integer. The actual value (a count of the number of records processed by each transaction instance) is provided when the transaction executes.

3.2 Defining Transactions in ARM 3.0

In ARM 2.0 the application provides the application and transaction names and the agent dynamically generates a transaction ID. In ARM 3.0 the application provides all three values.

The format of the application and transaction names has not changed. The format of the transaction ID has changed. Instead of the 4-byte transaction ID in ARM 2.0 the transaction ID is a 16-byte value. The 16-byte value is known in the ARM specification as the UTID (Universal Transaction Identifier).

The UTID is chosen when the application is developed, just like the application and transaction names. Once chosen, the UTID is a part of the application and will not change when the application is installed or activated.

There are no constraints on how the 16-byte identifier is formed but there is a recommendation. The recommended form is the UUID (Universal Unique Identifier) defined in the DCE RPC (Distributed Computing Environment Remote Procedure Call) standard [RPC]. In Microsoft Windows it is known as a GUID (Globally Unique Identifier). A UUID is formed from four components:

- The 6-byte IEEE 802.3 hardware address of a network interface card on the system (commonly known as the MAC address). If there is no 802.3
address on the system, some other identifier unique to the system is used.

- The current date and time.
- A forcibly incremented counter to handle high-frequency allocations.
- A clock sequence and related state to handle the retrograde movement of clocks (or unknown movement in the case of a reboot).

Given the way that a UUID is formed, there should never be a collision.

### 3.2.1 Registering Transaction Definitions

Just like ARM 1.0 and ARM 2.0, transactions must be defined before they can be measured. Applications define transactions with `arm_register_transaction()`, a new API call, shown in Figure 3. `arm_register_transaction()` replaces the paired `arm_init()` and `arm_getid()`. The UTID, application name, and transaction name are required parameters. The application-defined static attributes are optional.

![Figure 3. Registering ARM 3.0 Transactions](image)

An alternative registration mechanism may be part of the standard. The alternate mechanism would align with the DMTF’s Distributed Application Performance standard. At the time of writing this paper, a final decision had not been reached. The alternative mechanism is described in the “ARM and Standards” section.

### 3.3 Comparing ARM 2.0 and ARM 3.0

The primary advantage of the ARM 3.0 UTID is that there is a common identifier for a transaction type that does not change from system to system. This is important when correlators are being used to link client transactions to server transactions. In ARM 2.0 the correlator contains the dynamically assigned transaction identifier that can only be resolved to application and transaction names by requesting the agent that assigned the correlator to provide the mapping (or looking it up in a directory that the assigning agent previously updated). By using static identifiers, there is no need to request the mapping from the source agent.

### 4 Transaction Instances

An instance of a transaction is a transaction that is executing. If the application is an automated teller machine, and over the course of a day 584 people query their account balance, there are 584 instances of the “Query Account Balance” transaction.

#### 4.1 Transaction Instances in ARM 2.0

`arm_start()` marks the creation of a transaction instance. The lifetime of an instance is the time between the `arm_start()` and `arm_stop()` calls.

Each transaction instance is assigned a 32-bit handle. The handle is dynamically assigned by the agent as the return code from the `arm_start()` call. The handle is unique within a system at any point in time. The agent uses it to match up the `arm_stop()` call with its `arm_start()` call. Because this mechanism is used, ARM is thread-safe and process-safe.

Note that this transaction handle (for every instance) is different from the previously described transaction identifier, which is assigned only once per transaction type.

#### 4.2 Transaction Instances in ARM 3.0

The concept of a transaction instance is the same in ARM 2.0 and ARM 3.0. Neither has richer semantics. There are two changes in the mechanics.

- There is a new form for the transaction handle and new start, update, and stop API calls that use it.
- There is a new API call that can optionally be used instead of a start and stop pair.

#### 4.2.1 New Transaction Handle

An ARM 2.0 transaction instance is identified by a 4-byte integer handle. This handle is unique across all transaction types executing on the same system.

ARM 3.0 uses an 8-byte integer handle. In addition, the UTID is required to make the handle unique. This is because applications have the option of supplying the
transaction handle (or the agent can generate it). The UTID is needed to avoid collisions between different applications generating their own handles. In ARM 2.0 only the agent generates handles so the agent can insure that they are unique.

The new transaction handle form requires new API calls to distinguish them from ARM 2.0 calls. There are three new API calls:
- arm_start_transaction()
- arm_update_transaction()
- arm_stop_transaction().

The primary purpose of the UTID+handle combination is the same as the ARM 2.0 handle. It links arm_update_transaction() and arm_stop_transaction() calls to the corresponding arm_start_transaction() call.

The transaction handle can be provided in two ways:

- The agent assigns the handle. This is similar to ARM 2.0 except the handle is eight bytes instead of four (and it is passed by reference instead of by value).
- The application provides the handle, as shown in Figure 4. This could be particularly useful for applications that already have a unique internal identifier for each instance. If the identifier fits within eight bytes, the same value can be used, which simplifies programming and provides useful diagnostic information. The application is responsible for insuring that each instance of a transaction definition has a unique handle. Because no other application will have the same UTID, the combination results in a unique key.

4.2.2 Single Asynchronous Call per Transaction Instance

In ARM 2.0 the response time of transactions are measured by the agent. The response time is the elapsed time between the arm_start() and arm_stop(). The application is obliged to make the calls synchronously inline to have accurate measurements. This paradigm does not work or is not optimal in all cases. There are several reasons for this.

- The application may be within a critical section at the time the transaction starts and/or stops and the developer is concerned about calling an external library.
- The application may already time the transaction internally. It would be more efficient to provide this measurement with one call rather than making two calls.
- There might be concerns that making synchronous calls to an external library could disrupt carefully engineered performance. There might be no concern if the same program could make an asynchronous call from a background thread.

To address these reasons, ARM 3.0 adds a new call, arm_complete_transaction(), that is made once per transaction instance instead of separate start and stop calls. The call is shown in Figure 5. The application itself measures the response time and provides the value to the agent. The application also provides the timestamp when the transaction ended (from which the agent can deduce the start time by subtracting the response time). Knowing when the transaction executed could be useful for diagnostic purposes. The UTID for the transaction

![Figure 5. Single Asynchronous Call](image-url)
type must be registered first and status provided, as in any other instance.

Because the arm_complete_transaction() call contains both the response time and timestamp, it is not essential that the application make the call at the instant the transaction ends. The application could queue up these calls and send them all over in a batch, possibly from a background thread that runs at a low priority.

4.3 Comparing ARM 2.0 and ARM 3.0

There are several potential benefits of the new API calls. These will not apply to many application implementations, but with others the changes enable the use of ARM.

- Some or all handshaking between the application and the agent is eliminated.
- The application uses a statically defined UTID and does not have to manage identifiers dynamically assigned by the agent.
- The application has the option of providing its own handles with arm_start_transaction() and arm_stop_transaction().
- The application has the option of measuring the response time and providing the value, eliminating the need for synchronous calls.

5 Java Bindings for ARM

ARM 2.0 is defined using C language syntax. Because C is widely used, a program in practically any programming language can call a C program, and hence, a program in practically any programming language can call ARM. This is true also of Java applications (not applets) that use the Java Native Interface (JNI). JNI is the mechanism used by Java programs to load and call programs outside the Java Virtual Machine (JVM). The ARM Library is an example of such a program.

JNI is somewhat cumbersome to use because the call is not made directly from the Java program to the library. The call is made instead through two intermediate layers that provide the “escape hatch” from the constrained JVM environment. In spite of the extra work, there are many examples of companies using JNI with ARM. It would be better if programmers could call a Java interface directly.

ARM 3.0 defines native bindings for Java applications. The environment that has been prototyped is a mixed environment with both Java and non-Java programs. The expectation is that the same ARM Agent will support both types of programs. Each agent implementation would provide two libraries, one for Java programs and one for non-Java programs.

5.1 Solution Characteristics

The fundamental semantics of ARM apply to any transaction-oriented application. These semantics include defining transaction types within an application, and executing instances of these transaction types. The transaction instance has a status (Good, Failed, Aborted) and a response time. Optional metrics may be provided by the application. The semantics are the same in the Java version of ARM.

ARM 3.0 is the first version of ARM supporting native Java bindings. Because it is the first version, backwards compatibility is not an issue or requirement. Therefore Java bindings are defined using ARM 3.0 semantics.

The design reflects several other philosophies.

- The interface follows the paradigms and conventions of the language. Java programmers should feel the interface is natural and easy to use.
- The interface exploits language features peculiar to Java, such as garbage collection, inheritance, and overloaded constructors.
- The performance of Java and non-Java programs will be comparable. Java may be slower than native C, depending on the implementation of the compiler, but on the other hand, the Java version has some advantages, such as updating a metric once that is shared by any number of transaction instances.
- Java-based instrumentation is fully interoperable with non-Java instrumentation. This is most significant when correlators are passed between systems.

5.2 Solution Design

Figure 6 shows the object model of the solution. Figure 7 is a brief example of a program using Java/ARM. For the sake of clarity, some details have been omitted.

- ArmTranDefinition represents the definition of a transaction type. Creating and initializing an ArmTranDefinition object is equivalent to a call to arm_register_transaction().

- ArmTransaction represents a transaction instance. The methods ArmTransaction.start(), ArmTransaction.update(), ArmTransaction.stop()
and ArmTransaction.completeTransaction() are equivalent to their namesakes in the non-Java version of ARM 3.0.

- **ArmMetricDescription** represents the optional static attributes initialized in ARM 3.0 by calling arm_register_transaction(), except for the slot position (in which of the seven slots will this metric be found?). The slot position is specified in the ArmMetricGroup class. The metric description can be shared by one or several ArmTranDefinition objects.

- **ArmMetricXYZ** contains the actual values. “XYZ” is used as a condensed notation. The string “XYZ” is replaced with one of several strings, depending on the data type. There are ten classes, one for each data type supported by ARM. An example is ArmMetricCounter32. The same value can be shared among any number of ArmTransaction objects, from one or several ArmTranDefinition objects. An advantage of this capability is that updating one value in one object can effectively update a value in many transaction instances.

- **ArmMetricGroup** binds a transaction type (ArmTranDefinition) to metric descriptions and metrics. By using this class, the same metric value can appear in different slots with different transaction types, such as slot 4 on transaction type ABC and slot 1 on transaction type DEF.

The reader may wonder why slot assignments are used at all in the Java version. In the non-Java version of ARM, the slot definition needs to be specified because values are stored in the buffer at fixed positions. Java references eliminate the requirement for fixed positions, so why have slot assignments at all? The primary reason is to address a fringe case where instrumentation of the same transaction definition exists in both Java and non-Java versions. It would be important for a management application to be able to process the data in the same way, find the data into the same positions in records, and so on. It is also intuitive that the data show up in the positions defined in a configuration file, if one exists.

To avoid naming collisions, Java uses the concept of packages. Classes defined as part of a named package need unique names only within the scope of the package. A naming convention based on a company’s or organization’s domain name is in wide use. An example package name would be com.mycompany.mypackage. The CMG (Computer Measurement Group) has agreed to permit the Java/ARM package to be named within its domain, so the package name is org.cmg.arm.

### 5.3 Optimized Performance

Although Java programs are typically slower than C programs with the same logic, Java/ARM should not be significantly slower, and could turn out to be as fast or faster. Several characteristics of the design were specifically included to achieve optimal performance.

- Creation of objects is done under the control of the application and does not need to be done while the program is executing transactions. For example, although ArmTransaction represents a transaction instance, each object can be reused over and over.

![Figure 6. Java Object Model for ARM 3.0](image-url)
To readers familiar with a previous proposal for Java/ARM, this represents an improvement because the previous proposal created a new object every time a transaction instance executed.

- Data is not duplicated needlessly. For example, ArmTransaction might represent a transaction with several metrics defined. Instead of defining the metrics within the ArmTransaction class, which would require an update to every transaction instance object, a reference is used to a separate class. The same principle is used when defining any of the objects.

- Steps are taken within the interface or agent implementation to make data immutable (not changeable) when another object may be depending on its value not changing. By building the control into the infrastructure, constant rechecks of object state or redundant updates are avoided. In Figure 7 the myTranDef.register() method call changes the internal state of the ArmTranDefinition object so further updates are not allowed.

5.4 Java Applets – A Special Case

A Java application is just like a program in any other language. It can load and call other programs, it has access to system resources like the file system, and it can communicate over a network to any server (with appropriate authorization). A Java applet operates in a different environment than Java applications. Applets are intended to be downloadable across a network (such as the World Wide Web) and run on endpoint systems. Users are willing to do this only if they know that the applet cannot damage their system.

In order to make Java usable in this untrusted environment, applets execute under a set of constraints known as the “Java sandbox” [MACG98]. These constraints can only be overridden by taking extraordinary steps at the local system to give applets special privileges. Relevant constraints include:

- An applet cannot load and call a program on the local system.
- An applet cannot access local resources, such as memory, the file system, or execute I/O operations.
- An applet cannot initiate a network connection to any computer except the server from which it was loaded.

Even if applets are authorized to call local programs, such as the ARM Library, there is another problem. There isn’t in general any way to put the ARM Library on the client system where the application is being measured. The measurement must be done from within the applet. In other words, the applet must act as its own agent (by including some agent function) and it must communicate

```
//------------- Define a metric, allocate an object, & assign to a group --------------
ArmMetricDescription myDesc = new ArmMetricDescription(ARM_COUNTER32, "Counts transactions");
ArmMetricCounter32 myCounter = new ArmMetricCounter32();
ArmMetricGroup myGroup = new ArmMetricGroup();
myGroup.assign(3, myDesc, myCounter);

//--------------- Define the transaction type -------------------
ArmTranDefinition myTranDef = new ArmTranDefinition();
myTranDef.setTranId(myTranId);
myTranDef.setApplName(myApplName);
myTranDef.setTranName(myTranName);
myTranDef.setMetricGroup(myGroup);
myTranDef.register(); // This locks down the definition so it can’t be changed

//--------------- Create a transaction instance object -------------------
ArmTransaction myTran = new ArmTransaction(myTranDef);

//--------------- Execute the transaction ten times, updating the counter each time ------------
int recordCount;
for (int i=0; i<10; i++) {
    myTranDef.start();
    recordCount = processFile();
    myCounter.set(recordCount);
    myTranDef.stop();
}
```

Figure 7. ARM 3.0 Java Example
measurement data back to a server for processing. See [JOHN98] for a discussion of the issues.

The Java interface described in this paper has been prototyped for Java applications, but not for applets. Ideally the interface for Java applications and Java applets would be identical. The implementation of the agent supporting the interface will usually be different because most agents implementing ARM communicate across processes using native mechanisms, something the Java sandbox does not allow.

An agent for use in applets may require a different interface. For example, additional information may be needed before the agent can successfully communicate back to a server system through firewalls, proxies, and the like. These issues have not been addressed in the current prototype. So even if the ARM part of the interface turns out to be the same for Java applications and applets, a class within a Java applet that implements the Java/ARM interface may require additional configuration information.

6 ARM and Standards

The original version of ARM was developed jointly by Hewlett Packard and Tivoli Systems, and released in June 1996. Since then, three standards groups have participated in or been influenced by the development of ARM.

6.1 The ARM Working Group

ARM 2.0 and ARM 3.0 were jointly developed by an industry consortium, the ARM Working Group. Participating companies included Appliant, BMC, Boeing, Candle, Charles Schwab, Citicorp, Compuware, Hewlett-Packard, IBM, Landmark, Oracle, SAS, SES, Sun, Tivoli, Unify, and Wells Fargo. The ARM 2.0 software developer’s kit was released in December.

This paper describes ARM 3.0. The technical specification is essentially complete. The target date to release a final specification and a software developer’s kit for ARM 3.0 is early 2000.

6.2 The Open Group

The Open Group approved ARM 2.0 as a technical standard in July 1998 as part of its IT DialTone™ initiative [TOG98]. Work is underway in The Open Group on a reference implementation, test suite capability, branding, marketing, deployment and procurement.

6.3 Distributed Management Task Force

The DMTF (Distributed Management Task Force) [DMTF98], formerly the Desktop Management Task Force, is defining several standards for managing systems, networks, databases, and applications. In 1998 a new subgroup, DAP (Distributed Application Performance), was formed to address requirements for managing application performance. DAP is under the CIM (Common Information Model) [CIM98] umbrella. Most participants in DAP are also members of the ARM Working Group.

DAP has developed a conceptual model for application performance. The model addresses the runtime performance of transactions (called units of work in DAP). Unlike ARM, the DAP standard is not a programming API for instrumentation. Given their different roles, DAP is not an alternative to ARM, but rather a complement.

Except for the naming differences, the DAP model is the same as the ARM 3.0 model of a transaction.

- Units of work (ARM transactions) are defined with four static attributes.
  - A 16-byte key identifier (ARM UTID)
  - A runtime context (ARM application name)
  - A name (ARM transaction name)
  - Optional metrics (optional ARM static attributes)

- The measurements about units of work are response time and status.

- Optional attributes can be provided about any unit of work instance.

A MOF (Managed Object Format) file is an important part of all CIM standards. The MOF file is the template for defining the managed objects. The MOF file can also be modified to contain definitions of object instances, such as definitions of units of work. Because the ARM and DAP conceptual models are the same, it is possible to use a MOF file to register ARM transactions instead of making arm_register_transaction() calls. This is shown in Figure 8. Applications using ARM would be delivered with a MOF file. An agent supporting ARM 3.0 would be capable of importing the MOF file and initializing accordingly.

At the time this paper is being written, it has not been determined whether use of the MOF file would be part of the ARM 3.0 final specification (as mandatory for agents to support or as an option).

The use of a MOF file would provide another potential benefit. Previously it was asserted that the use of a static
UTID greatly simplifies correlation of client and server transactions across systems because the correlation application does not have to resolve a dynamically assigned transaction identifier means. The MOF file provides a way for an application written to use ARM 2.0 correlators to define a UTID for the pair of application and transaction names. This is done without any code changes whatsoever. An agent that imports the MOF file could then create correlators in an ARM 3.0 format, eliminating the need to dynamically resolve transaction identifiers.

7 Summary

This paper has described ARM and some of the factors that drove its creation. Extensions to ARM that will be part of ARM 3.0 were described. The rationale for the enhancements was discussed, as was relevant standards activities.

ARM 3.0 is expected to contribute to the increasing acceptance of ARM as a practical and useful standard for managing application performance.

8 References


[RPC] CAE Specification, C706, DCE 1.1: Remote Procedure Call