Open Group Technical Standard

Application Response Measurement
Issue 3.0 – Java Binding

The Open Group

Technical Standard

Application Response Measurement, Issue 3.0 – Java Binding

UK ISBN: 185912 232 9

US ISBN: 1931624 04 6

Document Number: C014

Published in the U.K. by The Open Group, October, 2001

Any comments relating to the material contained in this document may be submitted to The Open Group at:

The Open Group
Apex Plaza
Forbury Road
Reading
Berkshire, RG1 1AX
United Kingdom

or by electronic mail to

OGSpecs@opengroup.org

This specification has not been verified for avoidance of possible third party proprietary rights. In implementing this specification, usual procedures to ensure the respect of possible third party intellectual property rights should be followed.

Contents

Figures

Preface

The Open Group

.The Open Group is the leading vendor-neutral, international consortium for buyers and suppliers of technology. Its mission is to cause the development of a viable global information infrastructure that is ubiquitous, trusted, reliable, and as easy-to-use as the telephone. The essential functionality embedded in this infrastructure is what we term the IT DialTone. The Open Group creates an environment where all elements involved in technology development can cooperate to deliver less costly and more flexible IT solutions.

Formed in 1996 by the merger of the X/Open Company Ltd. (founded in 1984) and the Open Software Foundation (founded in 1988), The Open Group is supported by most of the world's largest user organizations, information systems vendors, and software suppliers. By combining the strengths of open systems specifications and a proven branding scheme with collaborative technology development and advanced research, The Open Group is well positioned to meet its new mission, as well as to assist user organizations, vendors, and suppliers in the development and implementation of products supporting the adoption and proliferation of systems which conform to standard specifications.

With more than 200 member companies, The Open Group helps the IT industry to advance technologically while managing the change caused by innovation. It does this by:

- Consolidating, prioritizing, and communicating customer requirements to vendors
- Conducting research and development with industry, academia, and government agencies to deliver innovation and economy through projects associated with its Research Institute
- Managing cost-effective development efforts that accelerate consistent multi-vendor deployment of technology in response to customer requirements
- Adopting, integrating, and publishing industry standard specifications that provide an essential set of blueprints for building open information systems and integrating new technology as it becomes available
- Licensing and promoting the Open Brand, represented by the ''X'' Device, that designates vendor products which conform to Open Group Product Standards
- Promoting the benefits of the IT DialTone to customers, vendors, and the public

The Open Group operates in all phases of the open systems technology lifecycle including innovation, market adoption, product development, and proliferation. Presently, it focuses on seven strategic areas: open systems application platform development, architecture, distributed systems management, interoperability, distributed computing environment, security, and the information

superhighway. The Open Group is also responsible for the management of the UNIX trademark on behalf of the industry.

Development of Product Standards

This process includes the identification of requirements for open systems and, now, the IT DialTone, development of Technical Standards (formerly CAE and Preliminary Specifications) through an industry consensus review and adoption procedure (in parallel with formal standards work), and the development of tests and conformance criteria.

This leads to the preparation of a Product Standard which is the name used for the documentation that records the conformance requirements (and other information) to which a vendor may register a product.

The ''X'' Device is used by vendors to demonstrate that their products conform to the relevant Product Standard. By use of the Open Brand they guarantee, through the Open Brand Trade Mark License Agreement (TMLA), to maintain their products in conformance with the Product Standard so that the product works, will continue to work, and that any problems will be fixed by the vendor.

Open Group Publications

The Open Group publishes a wide range of technical documentation, the main part of which is focused on development of Technical Standards and product documentation, but which also includes Guides, Snapshots, Technical Studies, Branding and Testing documentation, industry surveys, and business titles.

There are several types of specification:

- Technical Standards (formerly CAE Specifications)
  The Open Group Technical Standards form the basis for our Product Standards. These Standards are intended to be used widely within the industry for product development and procurement purposes.
  Anyone developing products that implement a Technical Standard can enjoy the benefits of a single, widely supported industry standard. Where appropriate, they can demonstrate product compliance through the Open Brand. Technical Standards are published as soon as they are developed, so enabling vendors to proceed with development of conformant products without delay.
- CAE Specifications
  CAE Specifications and Developers' Specifications published prior to January 1998 have the same status as Technical Standards (see above).
- Preliminary Specifications
  Preliminary Specifications have usually addressed an emerging area of technology and consequently are not yet supported by multiple sources of stable conformant implementations. They are published for the purpose of validation through implementation of products. A Preliminary Specification is as stable as can be achieved, through applying The Open Group's rigorous development and review procedures. Preliminary Specifications are analogous to the trial-use standards issued by formal

standards organizations, and developers are encouraged to develop products on the basis of them. However, experience through implementation work may result in significant (possibly upwardly incompatible) changes before its progression to becoming a Technical Standard. While the intent is to progress Preliminary Specifications to corresponding Technical Standards, the ability to do so depends on consensus among Open Group members..

- Consortium and Technology Specifications
  The Open Group publishes specifications on behalf of industry consortia. For example, it publishes the NMF SPIRIT procurement specifications on behalf of the Network Management Forum. It also publishes Technology Specifications relating to OSF/1, DCE, OSF/Motif, and CDE.
  Technology Specifications (formerly AES Specifications) are often candidates for consensus review, and may be adopted as Technical Standards, in which case the relevant Technology Specification is superseded by a Technical Standard..

In addition, The Open Group publishes:

- Product Documentation
  This includes product documentation—programmer's guides, user manuals, and so on—relating to the Pre-structured Technology Projects (PSTs), such as DCE and CDE. It also includes the Single UNIX Documentation, designed for use as common product documentation for the whole industry.
- Guides
  These provide information that is useful in the evaluation, procurement, development, or management of open systems, particularly those that relate to the Technical Standards or Preliminary Specifications. The Open Group Guides are advisory, not normative, and should not be referenced for purposes of specifying or claiming conformance to a Product Standard.
- Technical Studies
  Technical Studies present results of analyses performed on subjects of interest in area relevant to The Open Group's Technical Program. They are intended to communicate the findings to the outside world so as to stimulate discussion and activity in other bodies and the industry in general.

Versions and Issues of Specifications

As with all live documents, Technical Standards and Specifications require revision to align with new developments and associated international standards. To distinguish between revised specifications which are fully backwards compatible and those which are not:

- A new Version indicates there is no change to the definitive information contained in the previous publication of that title, but additions/extensions are included. As such, it replaces the previous publication.
- A new Issue indicates there is substantive change to the definitive information contained in the previous publication of that title, and there may also be additions/extensions. As such, both previous and new documents are maintained as current publications.

Corrigenda

Readers should note that Corrigenda may apply to any publication. Corrigenda information is published on the World-Wide Web at http://www.opengroup.org/corrigenda.

Ordering Information

Full catalogue and ordering information on all Open Group publications is available on the World-Wide Web at http://www.opengroup.org/pubs.

This Document

This document is a Technical Standard (see above)

Typographical Conventions

The following typographical conventions are used throughout this document:

- **Bold** font is used in text for options to commands, filenames, keywords, type names, data structures and their members.
- *Italic* strings are used for emphasis or to identify the first instance of a word requiring definition. Italics in text also denote:
  - command operands, command option-arguments or variable names, for example, substitutable argument prototypes
  - environment variables, which are also shown in capitals
  - utility names
  - external variables, such as *errno*
  - functions; these are shown as follows: *name*(). Names without parentheses are C external variables, C function family names, utility names, command operands or command option-arguments.
- Normal font is used for the names of constants and literals.
- The notation <file.h> indicates a header file.
- Names surrounded by braces, for example, {ARG_MAX}, represent symbolic limits or configuration values which may be declared in appropriate headers by means of the C `#define` construct.
- The notation [ABCD] is used to identify a return value ABCD, including if this is an error value.
- Syntax, code examples and user input in interactive examples are shown in `fixed width` font. Brackets shown in this font, `[]`, are part of the syntax and do not indicate optional items. In syntax the `|` symbol is used to separate alternatives, and ellipses (...) are used to show that additional arguments are optional.

Referenced Documents

The following documents have been used for reference during the creation of this specification:

ARM 2.0

Technical Standard, July 1998, Systems Management: Application Response Measurement (ARM) API (ISBN: 1-85912-211-6), published by The Open Group.

Acknowledgments

The Open Group gratefully acknowledges the work of the ARM Working Group, under the sponsorship of the Computer Measurement Group (CMG) I developing the ARM 3.0 For Java Programs, Iinterface and Package Specification, which provdes the technical source material for this Open Group Technical Standard.

In particular, the work of Mark Johnson, Tivoli Systems, and Ron Carelli, Hewlett-Packard, is gratefully recognized.

Trade Marks

Motif®, OSF/1®, UNIX®, and the "X Device"® are registered trademarks and IT DialTone™ and The Open Group™ are trademarks of The Open Group in the U.S. and other countries.

Hewlett-Packard is a trademark or registered trademark of Hewlett-Packard Company in the United States and other countries.

Java is a trademark or registered trademark of Sun Microsystems, Inc. in the United States and other countries.

Tivoli is a trademark or registered trademark of International Business Machines Corporation in the United States and other countries.

Chapter 1

Introduction

1.1 What is ARM?

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 a standard for measuring service levels of single-system and distributed applications. 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.

1.2 How is ARM Used?

ARM is a means through which business applications and management applications cooperate to measure the response time and status of transactions executed by the business applications.

Applications using ARM define transactions that are meaningful within the application. Typical examples are transactions initiated by a user and transactions with servers. As shown in Figure 1, applications on clients and/or servers call ARM when transactions start and/or stop. The agent in turn communicates with management applications, as shown in Figure 2, which provide analysis and reporting of the data.

The management agent collects the status and response time, and optionally other measurements associated with the transaction. The business application, in conjunction with the agent, may also provide information to correlate parent and child transactions. For example, a transaction that is invoked on a client may drive a transaction on an application server, which in turn drives ten other transactions on other application and/or data servers. The transaction on the client would be the parent of the transaction on the application server, which in turn would be the parent of the ten other transactions.

From the application developer's perspective ARM is a set of interfaces that the application loads and calls. What happens to the data after it calls the interfaces is not the developer's concern.

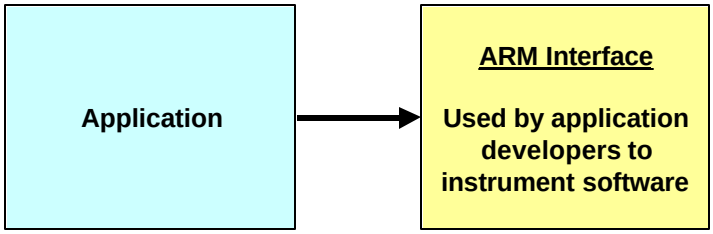

Figure 1. Application – ARM Interface

From the system administrator's perspective, ARM consists of the interfaces that applications load and call the classes that implement these interfaces, plus programs to process the data, as shown in Figure 2. How the data is processed is not part of the ARM standard, but it is, of course, important to the system administrator.

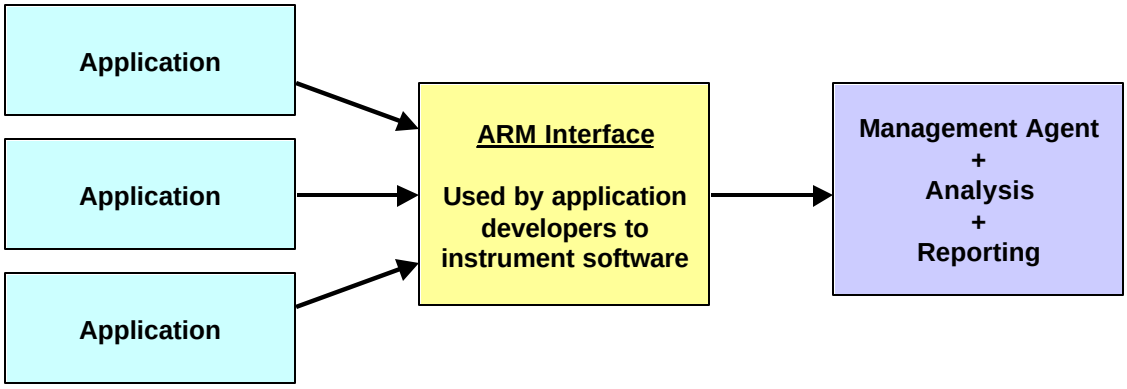

Figure 2. Application – ARM – Management System Interaction

1.3 What Transactions Should be Measured?

ARM is designed to measure a unit of work, such as a business transaction, or a major component of a business transaction, that is performance sensitive. These transactions should be something that needs to be measured, monitored, and for which corrective action can be taken if the performance is determined to be too slow.

Some questions to ask that aid in selecting which transactions to measure are:

- What unit of work does this transaction define?
- Are the transaction counts and / or response times important?
- Who will use this information?
- If performance of this transaction is too slow, what corrective actions will be taken?

1.4 The Evolution of ARM

ARM 1.0 was developed by Tivoli™ and Hewlett-Packard™ and released in June 1996. It provides a means to measure the response time and status of transactions. The interface is in the C programming language.

ARM 2.0 was developed by the ARM Working Group in 1997. The ARM Working Group was a consortium of vendors and end-users interested in promoting and advancing ARM. ARM 2.0 was approved as a standard of The Open Group™ (see [ARM 2.0]) in July 1998, part of the IT DialTone™ initiative. ARM 2.0 added the ability to correlate parent and child transactions, and to collect other measurements associated with the transactions, such as the number of records processed. The interface is in the C programming language.

ARM 3.0 is a new version of the standard that adds new capabilities. To date the specified interface is for the Java™ programming language. A C version has been mostly specified, but no prototype has been created. Java itself is evolving and there have now been at least three major versions of the language. The version that first became widely implemented and used was developed using the Java Development Kit (JDK) 1.1. New features were added in 1.2 and 1.3. For the most part a 1.1 program will work unchanged with a 1.2 or 1.3 JDK and programs. ARM 3.0 for Java Programs uses the JDK 1.1 version of the standard, and should be compatible with all later levels.

1.5 ARM 3.0 for Java Programs Overview

This specification describes four Java packages, Each package is equivalent to the block titled Arm Interface in Figure 1 and Figure 2.

- **org.opengroup.arm3.transaction** is the primary package that most applications use. The application calls a method when a transaction begins and ends, and the ARM implementation measures the response time.
- **org.opengroup.arm3.tranreport** is an alternative package that can be used by applications that measure the response time of their own transactions, and report the measurements after the fact.
- **org.opengroup.arm3.metric** can be used in addition to the org.opengroup.arm3.transaction package to report additional measurements about each transaction, such as a count of the amount of work accomplished.
- **org.opengroup.arm3.definition** can be used to report the binding of meta data to 16-byte binary identifiers.

An ARM implementation contains concrete classes that implement these interfaces. An ARM implementation may also be known as a “Management Agent” (and would be the part of the block labeled Management Agent + Analysis + Reporting that collects data). It is expected that companies will produce commercial ARM implementations, as was done for ARM 1.0 and ARM 2.0.

Business applications use ARM by creating objects that implement the interfaces in the packages, and then executing methods of the objects. The implementation of the classes takes care of all processing of the data, including moving the data outside the thread or JVM process to be analyzed and reported. The package is similar to the current ARM 2.0 interface, and a previously proposed ARM 3.0 interface.

Chapter 2

Using the ARM 3.0 Java Binding

2.1 An Example

An application uses ARM by creating objects that implement the ArmTransaction or ArmTranReport interfaces, and then invoking methods on these objects. A factory interface is provided to create the objects. The implementation of the ArmTransaction and ArmTranReport interfaces, which wll generally be provided by software vendors, process the measurement data transparently to the application. Here is an example:

```
// This snippet assumes these are already created:
ArmTransactionFactory tranFactory;
byte[] uuidBytesQueryBalance;
BankAccount myAccount;

// At initialization:
ArmUUID uuidQueryBalance;
ArmTransaction queryBalance;
uuidQueryBalance = tranFactory.newArmUUID(uuidBytesQueryBalance);
queryBalance     = tranFactory.newArmTransaction(uuidQueryBalance);

// At runtime as transactions execute
queryBalance.start();
   // The following line is the real job of the application
   status = myAccount.queryBalance(myCredentials);
queryBalance.stop(ARM_GOOD);
```

Many applications need no more. For a little more work, and a great deal more value, an application can link related transactions together. For example, a client transaction can be linked to an application server transaction, and the application server transaction can be linked to data server transactions. Understanding these relationships is tremendously useful for problem diagnosis, performance tuning, and capacity modeling. It also provides the means to link transactions to business transactions. Here is an example of an application server, which both receives a correlation token from its parent, and passes a correlation token to its children. The additional code to handle the correlation tokens is highlighted in boldface type.

```
// This snippet assumes these are already created:
ArmTransactionFactory tranFactory;
byte[] uuidBytesQueryBalance;
BankAccount myAccount;
```

```
byte[] parentBytes;   // correlation token from parent

// At initialization:
ArmUUID uuidQueryBalance;
ArmTransaction queryBalance;
uuidQueryBalance = tranFactory.newArmUUID(uuidBytesQueryBalance);
queryBalance     = tranFactory.newArmTransaction(uuidQueryBalance);

// At runtime as transactions execute
ArmCorrelator parent = tranFactory.newArmCorrelator(parentBytes);
queryBalance.start(parent);
ArmCorrelator corr = queryBalance.getCorr();
   // The following line is the real job of the application
   status = myAccount.queryBalance(myCredentials, corr);
queryBalance.stop(ARM_GOOD);
```

There are other optional features, but the examples above address the bulk of the requirements for using ARM. One feature enables applications to provide additional information about a transaction, such as a count of the work done (files processed, for example). Another feature enables an application to provide descriptive information about the transaction types, users, and data types.

Chapter 3

Programming Options

The application has two options for providing measurement data. In the first, and more widely used option, it calls an ARM transaction object just before and after a transaction executes, and the ARM transaction object makes the measurements. In the second option it makes all the measurements itself and reports the data some time later.

3.1 ARM Measures Transaction

Figure 3 shows the first, and most widely used, option. The object implementing ArmTransaction measures the response time. The application creates an instance of ArmTransaction. Immediately prior to starting a transaction, the application invokes start(). The ArmTransaction instance captures and saves the timestamp. Immediately after the transaction ends, the application calls the stop() method, passing the status as an argument. The ArmTransaction instance captures the stop time. The difference between the stop time and the start time is the response time of the transaction. As soon as the stop() method returns, the application is free to reuse the ArmTransaction instance. The data will have already been copied from it to be processed.

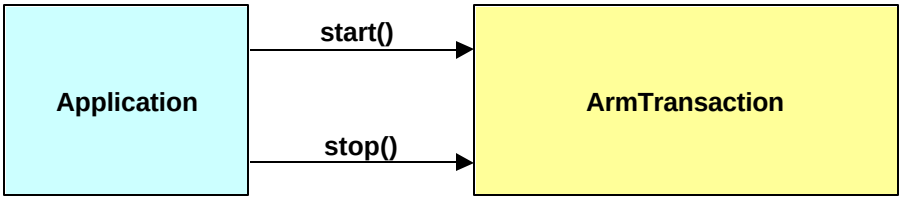

Figure 3. Measurement Using Start/Stop

The application optionally provides any number of heartbeat and progress indicators using update() between a start() and a stop(). This is shown in Figure 4. Heartbeats are useful for long-running transactions.

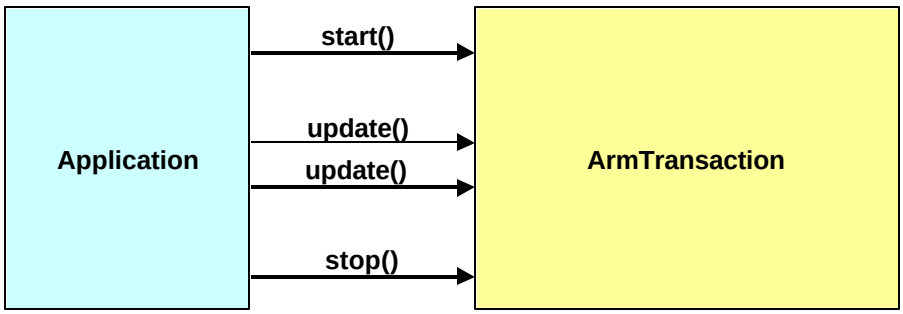

Figure 4. Application Using Heartbeats

3.2 Application Measures Transactions

Figure 5 shows the second option. The application itself measures the response time of the transaction. The application uses init() (or one of the variations) to identify the transaction by type and where it executed. After the transaction completes (the delay could be short or long), it populates an ArmTranReport object with data, and calls process() to initiate processing of the data. As soon as the process() method returns, the application is free to reuse the ArmTranReport instance. The data will have already been copied from it to be processed.

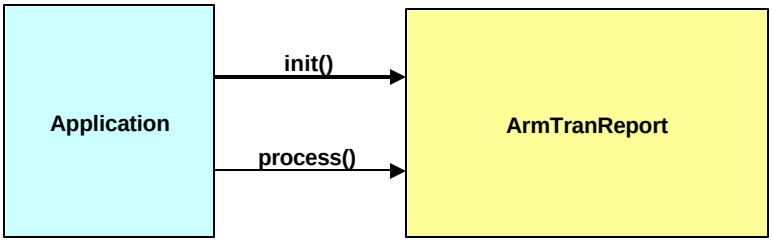

Figure 5. Measurement by the Application

3.3 Selecting Which Option to Use

In many situations the business application can use either programming option. In general, the recommendation is to use the first option (use ArmTransaction), unless that is not practical.

There is one situation for which the application must use Option 1:

- To provide heartbeats, the application must use the update() method of ArmTransaction between a start() and a stop(). Heartbeats are particularly valuable for long-running transactions. An ARM implementation may process updates, such as a real-time progress display, or check a threshold for a transaction that is taking too long.

There are two situations for which the application must use Option 2 (creating and populating ArmTranReport):

- Option 1 (ArmTransaction) requires that inline synchronous start() and stop() calls be used. The calls must be made at the moment the real transaction starts and stops. If they aren't, the timings will not be accurate. If the application finds this inconvenient or impractical, the application must use Option 2 (ArmTranReport). ArmTranReport can be used because the application provides both the response time and the stop time.
- If the transaction executes on system A but is reported to ARM on system B, Option 2 must be used for all the reasons stated above. In addition, the application provides additional information that identifies the system and JVM instance of the remote system where the transaction ran.

Chapter 4

Understanding the Relationships Between Transactions

There are several solutions available that measure transaction response times on one system, such as measuring the response time as seen by a client, or measuring how long a method on an application server takes to complete. ARM can be used for this purpose as well. This is useful data, but it doesn't provide insight into how transactions on servers are related to business transactions executed by users or other application programs. ARM provides a facility for correlating transactions within and across systems. This section describes how this is done.

4.1 A Typical Distributed Transaction

Most modern applications consist of programs distributed across multiple systems, processes, and threads. Figure 6 is an example.

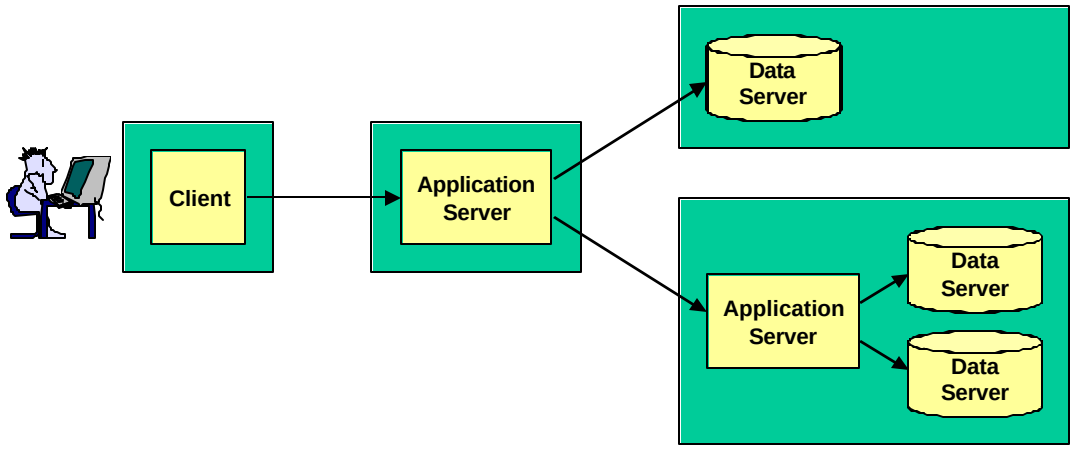

Figure 6. A Common Distributed Application Architecture

Figure 7 is an example transaction that runs on this application architecture. More correctly, Figure 7 shows a hierarchy of several transactions. To the user there is one transaction, but it is not unusual for the one transaction visible to the end-user to consist of tens or even over 100 sub-transactions.

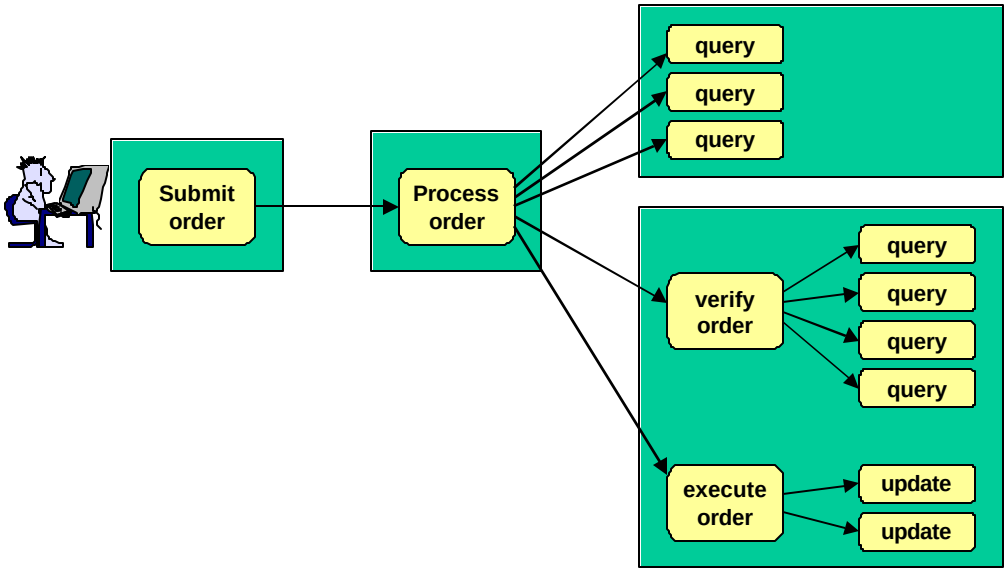

Figure 7. An Example of a Distributed Transaction

In ARM each transaction instance is assigned a unique token, named in ARM parlance a "correlator". To the application a correlator appears as an opaque byte array. There actually is a well-defined format to a correlator, and management agents and applications that understand it can take advantage of the information in it to determine where and when a transaction executed, which can aid enormously in problem diagnosis. Figure 8 shows the same transaction hierarchy as Figure 7, except that the descriptive names in Figure 7 have been replaced with identifiers. The lines are dotted instead of solid to indicate that without additional information, this would look to a management application like thirteen unrelated transactions.

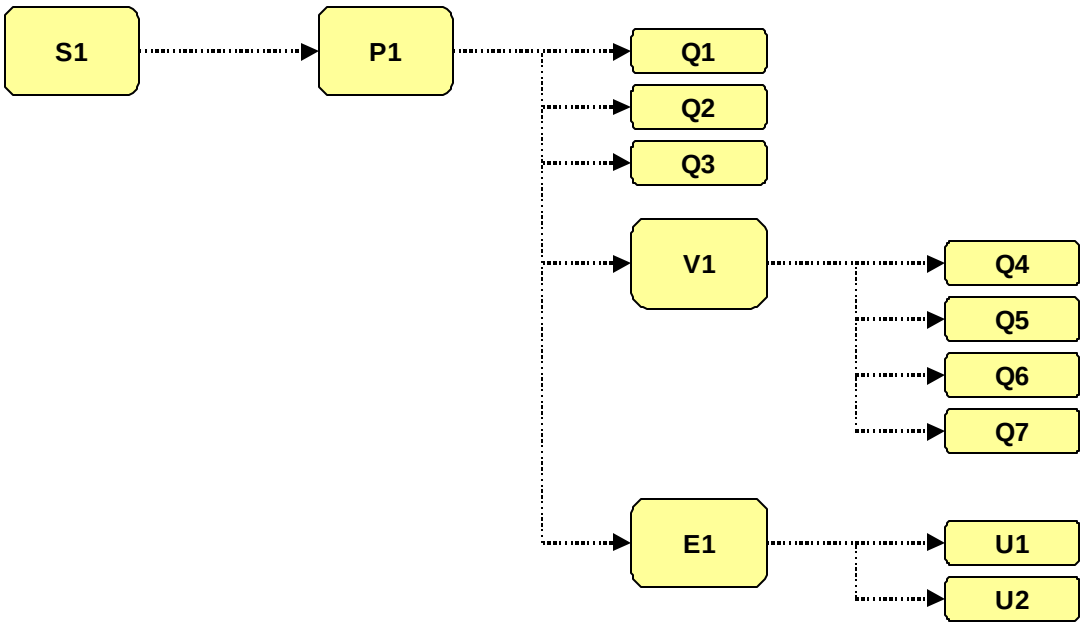

Figure 8. Distributed Transactions That Appear Unrelated

To relate the transactions together, the application components are each instrumented with ARM. In addition, each transaction passes the correlator that identifies itself to its children. In Figure 7 and Figure 8, the Submit Order transaction passes its correlator (S1) to its child, Process Order. Process Order passes its correlator (P1) to its five children – three queries, Verify Order, and Execute Order. Verify Order passes its correlator (V1) to its four children, and Execute Order passes its correlator (E1) to its two children.

The last piece in the puzzle is that each of the transactions instrumented with ARM passes its parent correlator to the ARM instrumentation class. The ARM instrumentation class knows the correlator of the current transaction. The correlators can be combined into a tuple of (parent correlator, correlator). Some of the tuples in Figure 8 are (S1,P1), (P1,Q1), (P1,E1), and (E1, U1). By putting the different tuples together, the management application can create the full calling hierarchy using the correlators to identify the transaction instances, as shown in Figure 9.

As an example of how this information could be used, if S1 failed, it would now be possible to determine that it failed because P1 failed, P1 failed because V1 failed, and V1 failed because Q6 failed.

Similar types of analysis could determine the source of response time problems. To analyze response time problems, additional information is needed. It's necessary to know if the child transactions execute serially, in parallel, or some combination of the two. The information may also be useful in locating unacceptable network latencies. For example, if the response time of S1 is substantially more than the response time of P1, and it is known that there is very little processing done on P1 that isn't accounted for in the measured response times, it suggests that there are unacceptable network or queuing delays between S1 and P1.

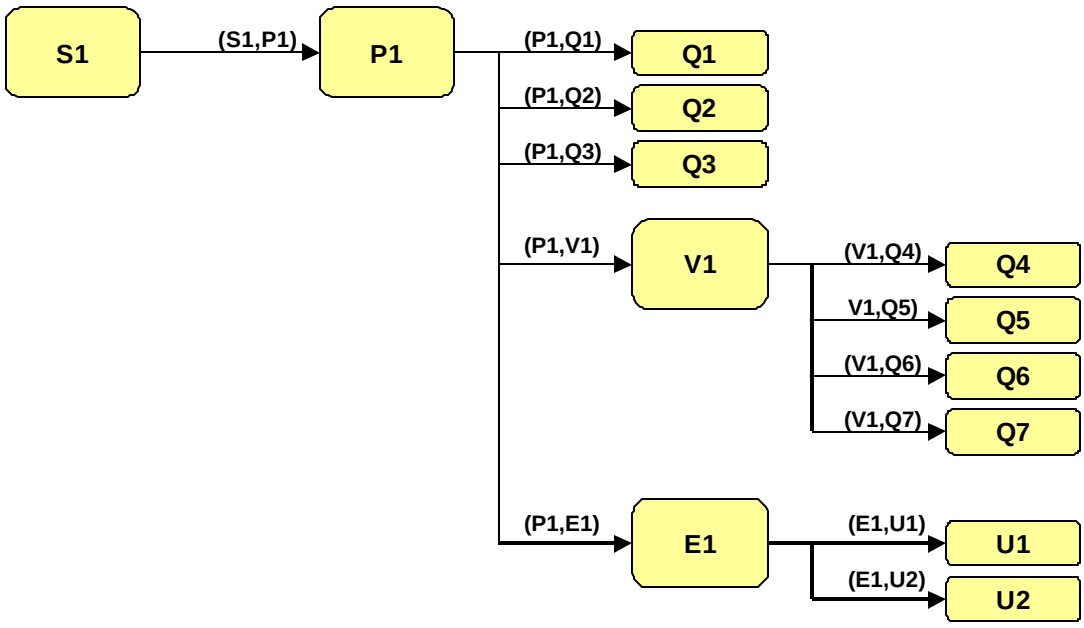

Figure 9. A Distributed Transaction Calling Hierarchy

Chapter 5

Additional Data About a Transaction

The identification information (transaction UUID) and measurement information (status, response time, stop time) for any transaction measured with ARM provides a great deal of value, and there may be no requirement to augment the information. However, there are situations in which additional information could be useful, such as:

- How "big" is a transaction? Knowing a backup operation took 47 seconds may not be sufficient to know if the performance was good. Additional information, such as the number of bytes or files backed up, provide much more meaning to the 47 seconds measurement.
- A transaction such as "get design drawings" may execute in less than a second for a simple part (e.g., a bracket). For complex parts, such as an engine, it may take many seconds to retrieve all the drawings, even if the system is performing well. Knowing the part number in this case makes the response time meaningful.
- The performance of a transaction will be affected by other workloads running on the same physical or logical system. Performance management tools may capture other information (e.g., CPU utilization) and combine it with response time measurements to plot the effect of CPU time on response time, which could be useful for planning the capacity of a system. However, other information that could be useful may not be available to performance management tools (e.g., the length of a queue internal to a program). It would be helpful for the application to provide this information.
- If a transaction fails it can be useful to know why. The required ARM status has four possible values: Good, Failed, Aborted, and Unknown. A detailed error code would be useful to understand why a transaction failed or was aborted. Capturing the code along with the other transaction information simplifies analysis by avoiding a later merge with, for example, error messages in a log file.

ARM provides a way for applications to provide these types of data. In ARM parlance the data are called "metrics". The use of metrics is *optional*.

ARM is not intended as a general-purpose interface for recording data. It is good practice to limit the use of metrics to data that is directly related to a transaction, and that helps to understand measurements about the transaction.

5.1 Data Categories

ARM supports ten data types. The data types are grouped in four categories. The categories are **counters**, **gauges**, **numeric IDs**, and **strings**.

5.1.1 Counters

A counter is a monotonically increasing non-negative value up to its maximum possible value, at which point it wraps around to zero and starts again. This is the IETF (Internet Engineering Task Force) definition of a counter.

A counter should be used when it makes sense to sum up the values over an interval. Examples are bytes printed and records written. The values can also be averaged, maximums and minimums (per transaction) can be calculated, and other kinds of statistical calculations can be performed.

ARM supports three counter types:

- 32-bit integer: ArmMetricCounter32
- 64-bit integer: ArmMetricCounter64
- 32-bit floating point: ArmMetricCounterFloat32. The floating-point standard is IEEE 754 (the same as the Java language).

5.1.2 Gauges

A gauge value can go up and down, and it can be positive or negative. This is the IETF definition of a gauge.

A gauge should be used instead of a counter when it is not meaningful to sum up the values over an interval. An example is the amount of memory used. If one measures the amount of memory used over 20 transactions in an interval and the average usage for each of these transactions was 15 MB, it does not make sense to say that 20*15=300 MB of memory were used over the interval. It would make sense to say that the average was 15 MB, that the median was 12 MB, and that the standard deviation was 8 MB. The values can be averaged, maximums and minimums per transaction calculated, and other kinds of statistical calculations performed.

ARM supports three gauge types:

- 32-bit integer: ArmMetricGauge32
- 64-bit integer: ArmMetricGauge64
- 32-bit floating point: ArmMetricGaugeFloat32. The floating-point standard is IEEE 754 (the same as the Java language).

5.1.3 Numeric IDs

A numeric ID is a numeric value that is used as an identifier, and not as a measurement value. Examples are message numbers and error codes.

Numeric IDs are classified as non-calculable because it doesn't make sense to perform arithmetic with them. For example, the mean of the last seven message numbers would hardly ever provide useful information. By using a data type of numeric ID instead of a gauge or counter, the application indicates that arithmetic with the numbers is probably nonsensical. An agent could create statistical summaries based on these values, such as generating a frequency histogram by part number or error number.

ARM supports two numeric ID types:

- 32-bit integer: ArmMetricNumericId32
- 64-bit integer: ArmMetricNumericId64

5.1.4 Strings

A string is used in the same way that a numeric ID is used. It is an identifier, not a measurement value. Examples are part numbers, names, and messages.

The strings are in standard 16-bit Unicode (UCS-2) characters (the same as the Java language).

ARM supports two string types:

- Strings of 1-8 characters: ArmMetricString8
- Strings of 1-32 characters: ArmMetricString32

5.2 How to Provide the Additional Data

The application provides the values in one of two ways, depending on how the transaction data are measured.

5.2.1 Using ArmTransactionWithMetrics

If the application is calling ArmTransaction start() and stop(), it creates instances of subclasses of ArmMetric (e.g., ArmMetricCounter32) and binds an instance to an ArmTransactionWithMetrics instance using ArmMetricGroup (not shown). Each ArmMetric subclass supports the set() method. Figure 10 shows this process. ArmTransactionWithMetrics is a subclass of ArmTransaction, and hence, implements all the methods of ArmTransaction, in addition to some methods for manipulating metrics.

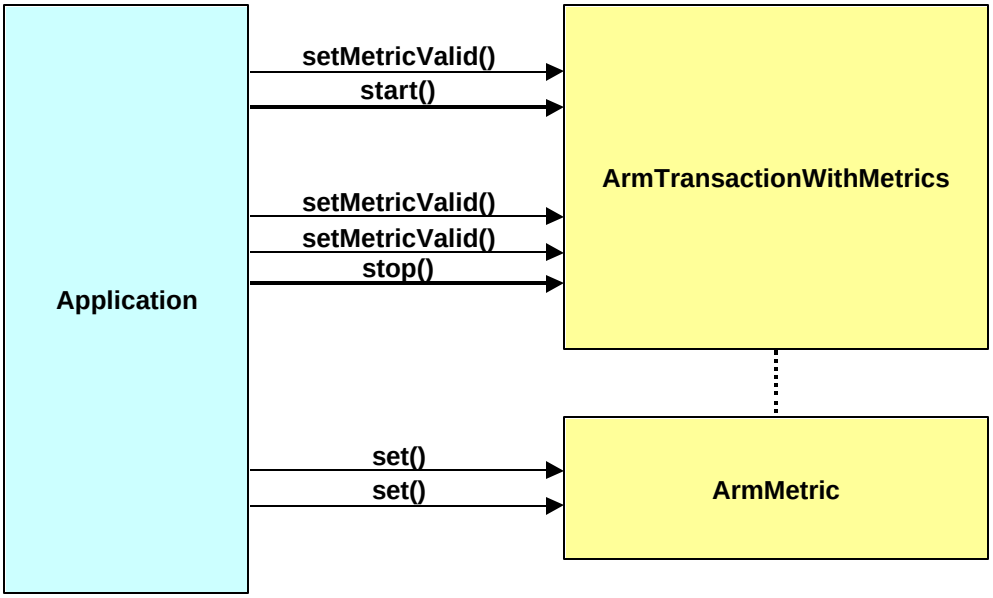

Figure 10. Providing Additional Data Using ArmTransaction and ArmMetric

Prior to calling start(), update(), or stop(), the application may set the value in each metric. The ArmTransactionWithMetrics method setMetricValid() is used to indicate if the data are valid. This is needed because the data might be valid only when stop() is executed, as an example. Figure 10 shows this process.

5.2.2 Using ArmTranReportWithMetrics

If the application is populating ArmTranReport instances and calling the process() method, the metric values are provided the same way, using a subclass, ArmTranReportWithMetrics. Prior to calling process(), setMetric() methods are called, such as setMetricCounter32(). This is illustrated in Figure 11. Whatever values are set when process() is executed are the values used for this transaction instance. If a value no longer contains meaningful data, clearMetric() is called prior to calling process().

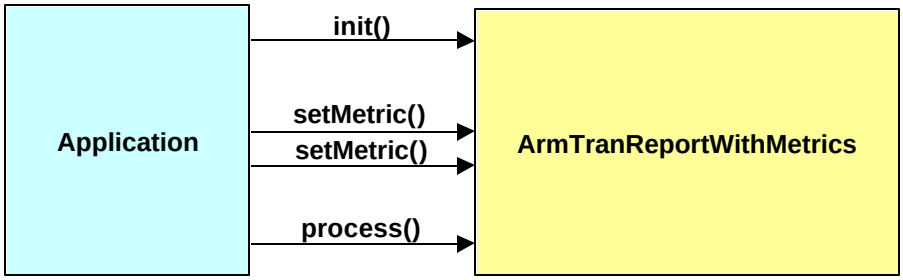

Figure 11. Providing Additional Data Using ArmTranReport

5.3 Processing Multiple Values of the Same Metric

Additional semantics are defined when using ArmTransactionWithMetrics in order to eliminate ambiguity. The ambiguity arises because the metric may be valid on some or all of the start(), update(), and stop() method calls. The following sections describe the semantics for each of the data type categories.

5.3.1 Counters

If a counter is used, its initial value must be set at the time of the `start()` call. The difference between the value when the `start()` executes and when `stop()` executes (or the value in the last `update()` call if no metric value is passed in `stop()`) is the value attributed to this transaction. Similarly, the difference between successive `update()` calls, or from the `start()` to the first `update()` call, or from the last `update()` to the `stop()` call, equals the value for the time period between the calls.

Here are three examples of how a counter would probably be used:

- The counter is set to zero at `start()` and to some value at `stop()` (or the last `update()` call). In this case, the application probably measured the value for this transaction and provided that value in the `stop()` call. The application always sets the value to zero at the `start()` call so the value at `stop()` reflects both the difference from the `start()` value and the absolute value.
- The counter is x1 at `start()`, x2 at its `stop()`, x2 at the next `start()`, and x3 at its `stop()`. In this case, the application is probably keeping a rolling counter. Perhaps this is a server application that counts the total workload. The application simply takes a snapshot of the counter at the start of a transaction and another snapshot at the end of the transaction. The agent determines the difference attributed to this transaction.
- The counter is x1 at `start()`, x2 at `stop()`, x3 (not equal to x2) at the next `start()`, and x4 at `stop()`. In this case, the application is probably keeping a rolling counter as in the previous example. But in this case the measurement represents a value affected by other users or transaction classes, so the value often changes from one `stop()` to the next `start()` for the same transaction class.

5.3.2 Gauges

Gauges can be set before `start()`, `update()`, and `stop()` calls. This creates the potential for different interpretations. If several values are provided for a transaction (one at `start()`, one at `update()`(s), and one at `stop()`), which one(s) should be used? In order to have consistent interpretation, the following conventions apply. Measurement agents are free to process the data in any way within these guidelines.

- The maximum value for a transaction will be the largest valid value passed at any time between and including the `start()` and `stop()` calls.
- The minimum value for a transaction will be the smallest valid value passed at any time between and including the `start()` and `stop()` calls.

- The mean value for a transaction will be the mean of all valid values passed at any time between and including the `start()` and `stop()` calls. All valid values will be weighted equally each time a `start()`, `update()`, or `stop()` executes.
- The median value for a transaction will be the median of all valid values passed at any time during the transaction. All valid values will be weighted equally each time a `start()`, `update()`, or `stop()` executes.
- The last value for a transaction will be the last valid value passed whenever any `start()`, `update()`, or `stop()` executes.

5.3.3 Numeric IDs

The last value passed when any of the `start()`, `update()`, or `stop()` calls are made will be the value attributed to the transaction instance. For example, if a value is valid at `start()` but not when any `update()` or `stop()` call executes, the value passed at the `start()` is used. If a value is valid when `start()` executes and when `stop()` executes, the value when `stop()` executes is the value for the transaction instance. This convention is identical to the String convention.

5.3.4 Strings

The last value passed when any of the `start()`, `update()`, or `stop()` calls are made will be the value attributed to the transaction instance. For example, if a value is valid at `start()` but not when any `update()` or `stop()` call executes, the value passed at the `start()` is used. If a value is valid when `start()` executes and when `stop()` executes, the value when `stop()` executes is the value for the transaction instance. This convention is identical to the Numeric ID convention.

Chapter 6

Creating ARM Objects

6.1 Overview of Java Interfaces

The ARM 3.0 for Java Programs standard defines Java interfaces. A Java interface is an abstract specification of method signatures. Following is an example of an interface. This interface is named `ArmMetricGroup`, it is part of the org.opengroup.arm3.metric package, and it defines three method signatures: `clear(int)`, `get(int)`, and `set(int,ArmMetric)`.

```
package org.opengroup.arm3.metric;
public interface ArmMetricGroup
{
   public void clear(int slot);
   public ArmMetric get(int slot);
   public void set(int slot, ArmMetric metric);
}
```

A program cannot create an instance (object) of an interface because there's no code to execute. Instead a program creates an instance of a concrete class that ***implements*** the interface. In the following two code fragments, each of which would be in its own file, two classes are defined (MyGroup and AnotherVendorsOne). Each class declares that it implements the ArmMetricGroup interface (and they import all the class and interface definitions in the org.opengroup.arm3.metric so the Java compiler can reconcile all the names). Each class includes method bodies for at least the three methods defined in ArmMetricGroup. If it doesn't, the Java compiler will generate an error. Other methods may also be included. In these examples, MyGroup has one other method (`privateStuff()`) and AnotherVendorsOne has two other methods (`differentStuff1()` and `differentStuff2()`).

```
import org.opengroup.arm3.metric.*;
public class MyGroup
       implements ArmMetricGroup
{
   public void clear(int slot)
      { // program code goes here }
   public ArmMetric get(int slot)
      { // program code goes here }
   public void set(int slot, ArmMetric metric)
      { // program code goes here }
   private void privateStuff()
      { // program code goes here }
}
```

```
import org.opengroup.arm3.metric.*;
public class AnotherVendorsOne
       implements ArmMetricGroup
{
   public void clear(int slot)
      { // program code goes here }
   public ArmMetric get(int slot)
      { // program code goes here }
   public void set(int slot, ArmMetric metric)
      { // program code goes here }
   private void differentStuff1()
      { // program code goes here }
   private void differentStuff2()
      { // program code goes here }
}
```

To create an object that implements the `ArmMetricGroup` interface, a program could create an instance of either `MyGroup` or `AnotherVendorsOne`. By assigning the object to a variable of type `ArmMetricGroup`, this variable can be used as if `ArmMetricGroup` is a concrete class. In the following code snippet, `group` is of type `MyGroup` and can execute any of the four methods of `MyGroup`. `g5` is of type `ArmMetricGroup` so it can execute only the three methods in `ArmMetricGroup`. The program statement "`g5.privateStuff()`" would generate a compiler error because `privateStuff()` is not defined in the `ArmMetricGroup` interface, whereas "`group.privateStuff()`" does not result in a compiler error.

```
MyGroup group = new MyGroup();
ArmMetricGroup g5 = (ArmMetricGroup) group;
int mySlot = 4;
g5.clear(mySlot);
group.clear(mySlot);
group.privateStuff();
```

6.2 Creating ARM Objects in an Application

The discussion up to now has been a short tutorial on Java interfaces and would apply to any Java program. The remainder of this section describes how applications (not applets) create objects that implement the ARM 3.0 for Java Programs interfaces. The next section describes how applets create the objects.

A fundamental characteristic of ARM is that an application that uses ARM will be able to work with any ARM implementation, whether written in-house or purchased from a vendor. Vendors compete with each other to provide better ARM implementations. The use of Java interfaces creates a potential problem because each vendor will have its own names for its own classes. In the examples above, `MyGroup` and `AnotherVendorsOne` are names that are not part of the standard. Further, a program that uses ARM should never use either name in a program because if it does, that program is restricted to only working with the ARM implementation from that particular vendor. But a program cannot create an instance of an object which implements an interface, such as the `ArmMetricGroup` interface, without naming a specific class. It is a compiler error to code "`ArmMetricGroup g5 = new ArmMetricGroup();`". So how can a program create a concrete class without naming the class directly?

ARM uses two mechanisms to create objects that implement the ARM interfaces. Together, these mechanisms permit a system administrator to choose an ARM implementation regardless of the class names of the implementation while allowing the application to work with any ARM implementation.

1. The ARM 3.0 for Java Programs standard defines four factory interfaces, one for each package. New objects are created by first creating an object that implements a factory interface, then invoking methods of the factory interface. The factory methods are used instead of using the Java "`new`" operator. The four factory interfaces are:

   ```
   ArmDefinitionFactory
   ArmMetricFactory
   ArmTranReportFactory
   ArmTransactionFactory
   ```

2. Using factory interfaces alone does not avoid naming the classes in each ARM implementation, because the objects implementing the factory interfaces need to be created by name. The application does not know the factory class names in advance (otherwise it would only work with one ARM implementation). The application gets the names of the factory classes through the use of the Java system properties. A system administrator assigns the names of the factory classes to the properties before starting an application that uses ARM.

The remainder of this section describes the process in more detail.

All JDKs implement the `java.lang.System` and `java.util.Properties` classes. These classes contain several methods to manipulate properties. `java.util.Properties` is a hash table containing properties. A property is a `String` and it is referenced within the hash table by a key, which is also a `String`. Java programs can create instances of `java.util.Properties` for their own purposes. `java.lang.System` creates a special instance of `java.util.Properties`

that is a singleton within the JVM. It provides a single place to store property values that will be available to all programs running within the JVM.

ARM defines four property keys, one for each of the four factory classes. Each factory interface defines a static (class) constant named "`propertyKey`". The value of each constant is the same name as the factory interface. For example, the `ArmDefinitionFactory` interface assigns the value `"ArmDefinitionFactory"` to its constant variable `propertyKey`. Each of the following statements is in the respective factory interfaces:

```
public static final String propertyKey = "ArmDefinitionFactory";
public static final String propertyKey = "ArmMetricFactory";
public static final String propertyKey = "ArmTranReportFactory";
public static final String propertyKey = "ArmTransactionFactory";
```

During initialization of the ARM environment in a JVM, a program provided by the system administrator will assign a class name in the system properties for each property key. For example, the following code snippet assigns the class name `"com.vendor1.arm.ArmTranFactory"` to the property key for `ArmTransactionFactory`. This would be repeated for the other three factory interfaces. In this case a company with a domain name of vendor1.com presumably supplies the ARM implementation.

```
Properties p = System.getProperties();
String valueTranFactoryClass = "com.vendor1.arm.ArmTranFactory";
String keyTranFactoryClass = ArmTransactionFactory.propertyKey;
p.put(keyTranFactoryClass, valueTranFactoryClass);
```

To create any of the ARM objects an application first creates an instance of the appropriate factory class. It then uses the methods of the factory class to create the objects that implement the ARM interfaces. It is common for an application to create one instance of each of the four factory classes during initialization, and then use them to create all the other objects. However, there is no requirement to do so – the application can create any number of instances of each factory, and can create one whenever it needs one.

In the following code snippet, `tranFactoryName` is the ***name*** of the factory class (for example, "`com.vendor1.arm.ArmTranFactory`"), `tranFactoryClass` is the factory ***class*** (all Java classes can be represented by an instance of `java.lang.Class`), and `tranFactory` is an ***instance of the factory*** class. `tranFactory` can be used to create any number of instances of `ArmTransaction`. Three are created in this example, all for the same transaction UUID.

```
Properties p = System.getProperties();
String keyTranFactoryClass = ArmTransactionFactory.propertyKey;
String tranFactoryName     = p.getProperty(keyTranFactoryClass);
Class  tranFactoryClass    = Class.forName(tranFactoryName);
ArmTransactionFactory tranFactory;
tranFactory = (ArmTransactionFactory) tranFactoryClass.newInstance();

byte[] bytesTranUUID;  //Assume this already has a value
ArmUUID tranUUID = tranFactory.newArmUUID(bytesTranUUID);
ArmTransaction tran1 = tranFactory.newArmTransaction(tranUUID);
ArmTransaction tran2 = tranFactory.newArmTransaction(tranUUID);
ArmTransaction tran3 = tranFactory.newArmTransaction(tranUUID);
```

6.2.1 Convenience Methods

ARM implementations or SDKs may (but are not required to) provide convenience routines that hide some of these details. A suggested way to do this would be to provide class methods to create the factories by embedding the logic from above. For example:

```
public class ArmFactory
{
 public static ArmDefinitionFactory createArmDefinitionFactory() {}
 public static ArmTransactionFactory createArmTransactionFactory() {}
 public static ArmTransReportFactory createArmTransReportFactory() {}
 public static ArmMetricFactory createArmMetricFactory() {}
}
```

6.3 Creating ARM Objects in an Applet

The approach of using the system properties to identify the names of the concrete classes works for Java applications but it will not work for Java applets. Applets do not have access to the system properties, except a few that are expressly permitted. This section describes how to provide the class names to applets.

The basic principles remain the same. The applet should not change even if the ARM implementation changes. A system administrator should control which ARM implementation is used by each applet. This will be the system administrator of the server from which the applet is loaded.

The Java language permits applets to access files on the server system from which they originated, as long as those files are in the applet's codebase. By default the codebase is the directory that contains the HTML file that loaded the applet. The HTML file can specify the codebase to be a different directory using the CODEBASE tag. Classes in the applet's unnamed package (any class that doesn't specify a package) are taken from the same directory as the codebase. For classes that are members of a package, the codebase is extended by the package name. For example, if the codebase is www.abc.com/test, the ARM package would be in directory www.abc.com/test/org/opengroup/arm3/transaction. An applet can get the URL of the codebase using the `getCodeBase()` method of the `java.applet` class.

ARM defines a similar mechanism for applets as for applications. The main difference is that an application gets its properties from the system properties, whereas an applet gets its properties from a file named `arm.properties`. `arm.properties` must reside in the codebase of the applet. An application uses `java.lang.System.getProperties()` to create an instance of the `java.util.Properties` class. An applet creates an instance of `java.util.Properties` by loading it from the data in `arm.properties`. The format of each property, and the keys that identify the four factory classes, are identical.

The following example shows how an applet would initialize its `Properties` object and create an instance of `ArmTransactionFactory`, then use the factory to create an instance of `ArmTransaction`.

```
// Retrieve properties file from codebase
URL urlCodeBase = getCodeBase();
URL urlArmProp = new URL(urlCodeBase.getProtocol(),
                     urlCodeBase.getHost(),
                     urlCodeBase.getPort(),
                     urlCodeBase.getFile()
                     +"arm.properties");
InputStream in = urlArmProp.openStream();
Properties armProp = new Properties();
armProp.load(in);

// Get the factory class name and create an instance
String keyTranFactoryClass = ArmTransactionFactory.propertyKey;
String tranFactoryName = armProp.getProperty(keyTranFactoryClass);
Class  tranFactoryClass = Class.forName(tranFactoryName);
```

```
ArmTransactionFactory tranFactory;
tranFactory = (ArmTransactionFactory) tranFactoryClass.newInstance();

// Create an ArmTransaction instance
byte[] bytesTranUUID;
ArmUUID tranUUID = tranFactory.newArmUUID(bytesTranUUID);
ArmTransaction tran = tranFactory.newArmTransaction(tranUUID);
```

One final restriction, imposed by the Java language on applets, is that the implementation of the interface, i.e., the concrete classes that implement the ARM interfaces, must also reside in the applet's codebase.

Chapter 7

Providing Descriptive Information

7.1 Differences between ARM 3.0 versus ARM 1.0/2.0

In ARM 1.0 and 2.0, descriptive information about transactions and metrics were provided as UTF-8 character strings through the API. The application and user names were provided with the arm_init() call. The transaction names were provided with the arm_getid() call. In response to the combination of the names, the ARM library provided by the management agent supplied a 32-bit transaction ID that was unique within that system. This ID varied from system to system, and would generally change each time a system or an ARM implementation was reinitialized. The ID represented a dynamic binding, much like a file handle in the C programming language. These mechanisms required substantial handshaking between the application and the agent, which in some cases was problematic.

Two key objectives of ARM 3.0 are to improve the separation between identification and descriptive information, and to make the programming interfaces more loosely coupled. This is accomplished by referring to transactions, metrics, and users using unique 16-byte identifiers. Applications pass the 16-byte identifiers to fully and precisely identify which transaction type and metrics are being used. This also makes handling of different languages easier and intuitive.

7.2 Providing Descriptive Information through the Java Interfaces

To give meaning to the 16-byte identifiers, applications have the *option* of providing the name(s) and any other information, such as the format of a metric, that is associated with each 16-byte identifier. Why this is an option will be shown below. When provided, the ArmTranDefinition, ArmMetricDefinition, and ArmUserDefinition classes are used, as shown in Figure 12.

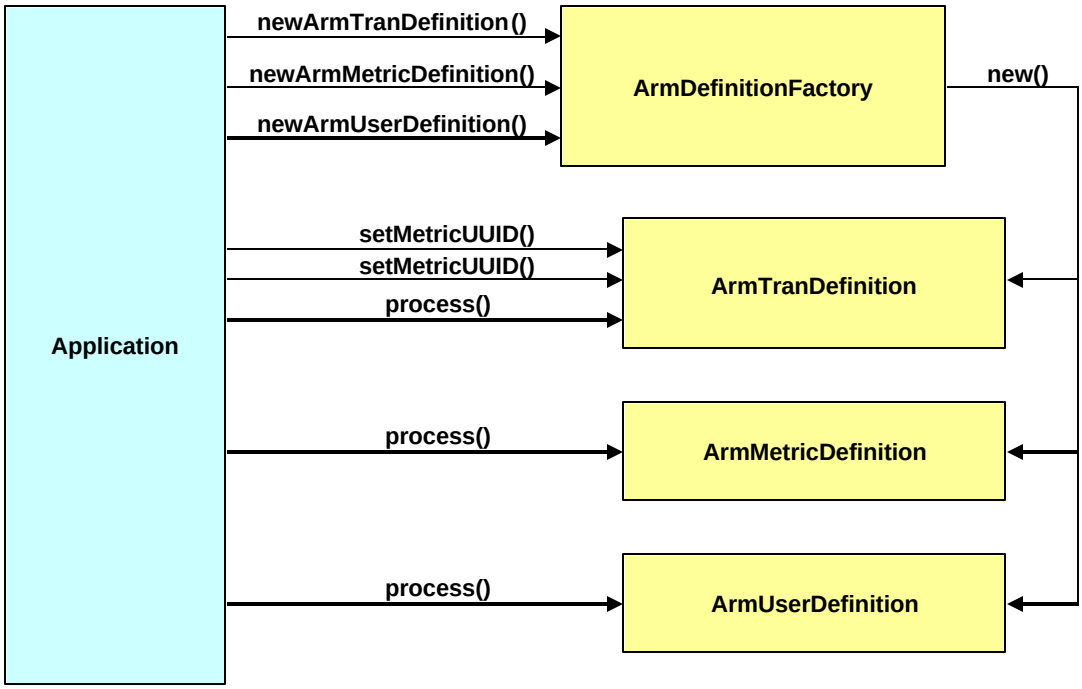

Figure 12. Providing Descriptive Information Through the Application Interface

Because using definition objects is optional, the only identification data used and available when measurements are collected are the 16-byte identifiers. An example of this is shown in Figure 13, where each row in the table corresponds to a transaction that has executed, and shortened identifiers have been used for brevity.

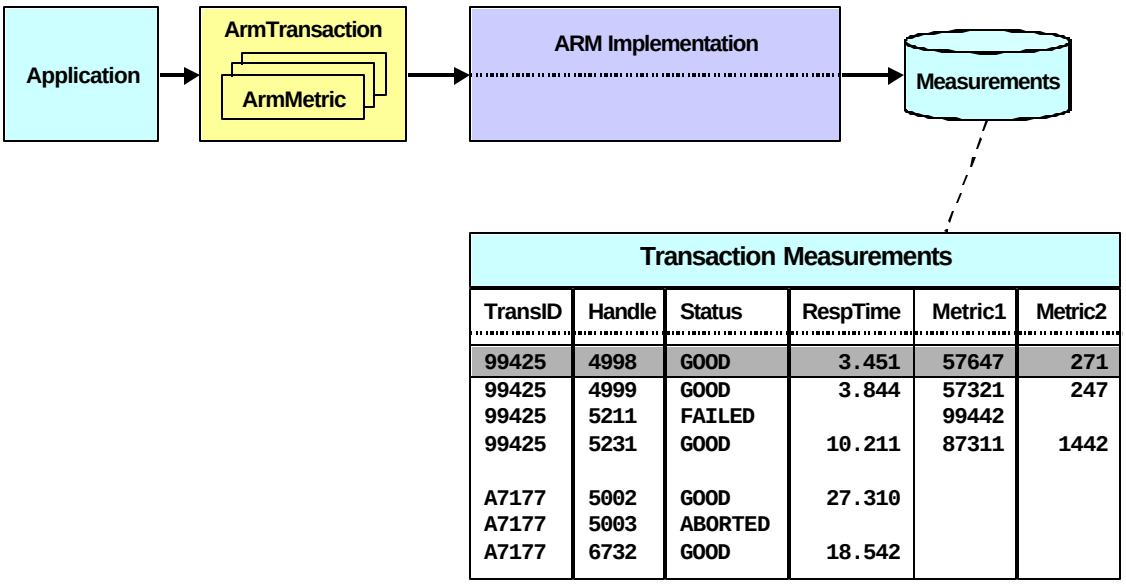

Figure 13. Providing Measurements Without Any Descriptive Information

The data in Figure 13 provide the status and response time for transaction IDs "99425" and "A7177" (shorthand for the 16-byte identifiers) and the values for metric IDs "Metric1" and "Metric2". To be useful, the IDs need to be translated into names that an analyst would understand. Figure 14 is an example that provides the descriptive information needed to understand the measurements.

The descriptive information is defined in two tables. The "TransID" field in the Transaction Measurements table indexes a row in the TransID column of the Transaction Definitions table. This provides the names of the application and transaction. It also provides the metric IDs. The metric IDs index the MetricID column of the Metrics Definitions table, from which it is apparent the metrics are named "Stock on Hand" and "Part Number" Using the information, the complete human-understandable information shown in the "EXAMPLE" box in the upper right of Figure 14 can be derived.

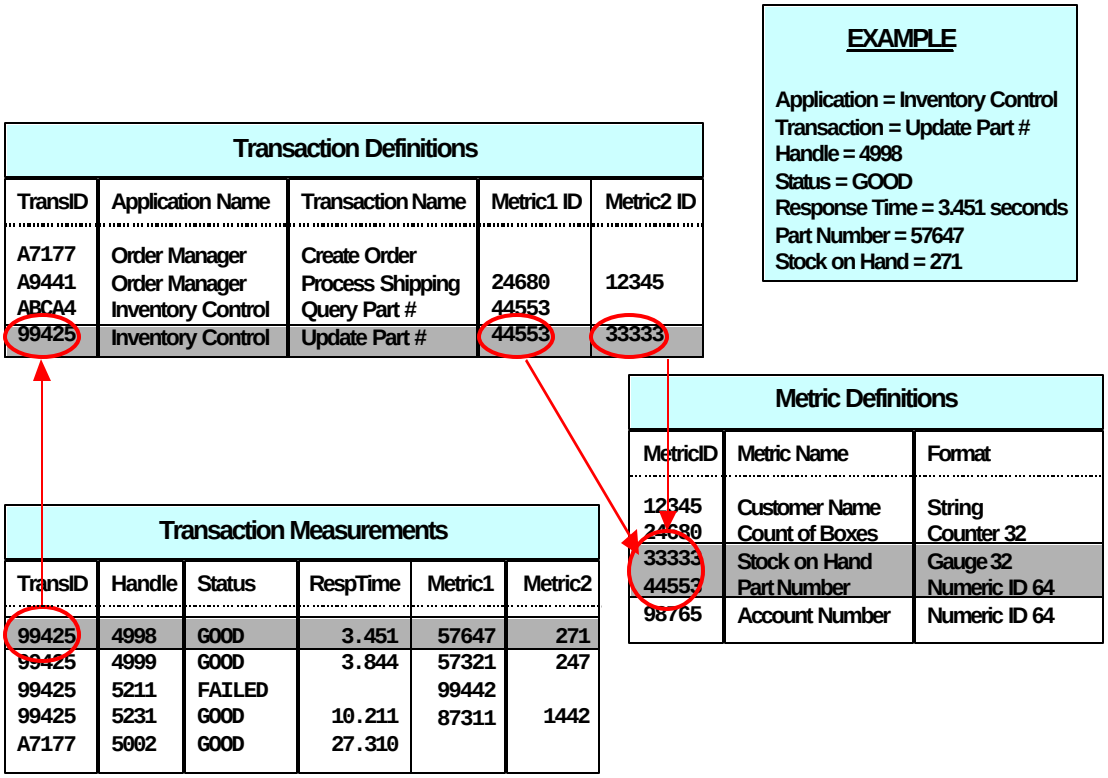

Figure 14. Combining Ids and Handles With Descriptive Information

These tables could be created from the information in ArmTranDefinition and ArmMetricDefinition. Figure 15 shows this process. The measurement data and descriptive information is kept separate until the reporting step. This is true even if the data all flows through the same ARM implementation. The data are treated as logically separate and not combined until it is time to report on the data.

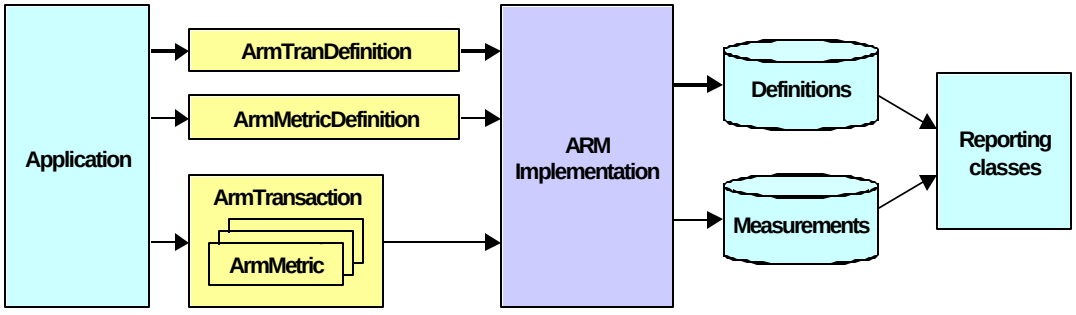

Figure 15. Providing Descriptive Information Through the ARM Application Interface

7.3 Providing Descriptive Information Offline or Out-of-band

Because the descriptive information is not combined with the measurement data until the data are reported, there is no requirement to flow the definitional data through the application interfaces. Providing facilities to flow the descriptive information data through the application interfaces is offered as a possibly convenient option to the application. However, there are applications that, because of their structure, do not really have an initialization stage that can be used to provide the definitional data at runtime. For these types of applications, or any application preferring to use an offline method, the information can be imported in an offline batch mode. The applications provide only the measurement information and the UUIDs through the application interfaces.

This alternative is shown in Figure 16. The "Arm Implementation" in the "Offline batch job" block will be implementation specific. The org.opengroup.arm3.application interfaces could be used. Other implementations might provide a script to load a text file containing the data, or to store the data in a CIM implementation, from which it could be extracted.

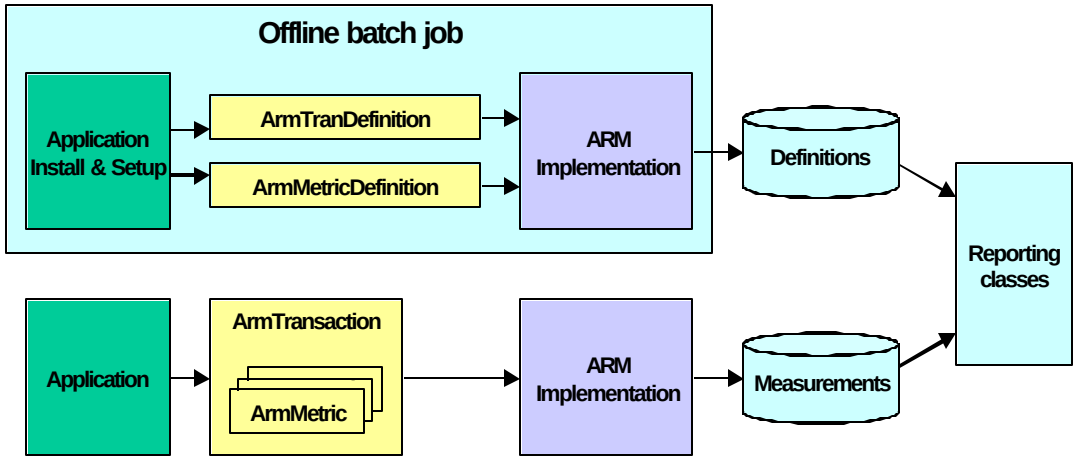

Figure 16. Providing Descriptive Information Offline

Chapter 8

Error Handling Philosophy

It is inevitable that errors will occur when using the ARM interface. The error handling philosophy of the ARM standard can be summed up as the following: *"Programmers and system administrators need to know about errors; programs do not."* The practical effect of this philosophy is that applications do not need to check for errors, except when creating factory classes, when exceptions could be thrown.

An application that contains programming errors, or that receives invalid data, could generate invalid measurement data. This is a problem that programmers and system administrators should correct. But at runtime there's nothing an application can do about it, so the ARM interface takes the approach of being as unobtrusive as possible, and permitting the application logic to flow normally. Programmers testing programs, and system administrators managing systems using ARM, should check for error reports from ARM implementations.

Any method that creates an ARM object, or a copy of an ARM object, will always return a valid non-null object of that type. If invalid data is provided, the data within that object may be incorrect or meaningless. However, the object will be syntactically correct, that is, it will be a valid Java object, and any of its methods can be invoked without causing an exception.

8.1 Errors the Application Does Not Need to Test For

None of the interfaces in the four packages that comprise the standard define exceptions, and none throw exceptions, except those that are so pervasive in Java classes that they do not have to be declared. Exceptions that do not have to be declared are those that are subclasses of java.lang.Error or java.lang.RuntimeException. Such exceptions can be thrown by practically any method. One RuntimeException that is often encountered is ArrayIndexOutOfBoundsException. Examples of exceptions that do have to be declared are IOExceptions.

Here are some examples of errors in the use of the ARM interface that are not exceptions:

- The application passes a null pointer when a non-null pointer is required.
- The application passes an invalid format ID or status value.
- The application passes an incorrectly formed correlator (which it may have received from the program that called it, in which case the problem is in the program or system that called it).
- When using ArmTransaction, two start() methods are executed consecutively without an intervening stop() or reset(), or an update() method is executed without a start() first being executed.

ARM has two principles for handling this type of error:

- The programmer (during program development) and/or the system administrator (after the application has been deployed into production) need to be aware of them so the problem can be corrected.
- An application running in production does not need to be aware of them because there's nothing the application can do about them. If there's a programming error that sets invalid data, the program will continue to set the invalid data. Telling the program that an error occurred provides no value and just clutters up the program. The program could write an error message to a log, but a drawback to doing so is that a system administrator would have to look in many places to find all the possible errors, because each application might write the data in a different place.

The recommended approach is for the ARM implementation to have a mechanism for providing programmers and/or system administrators with error notification. For example, the ARM implementation could write the data to a log file and/or create and send an error event to an event console. During development and testing, the programmer would inspect the log file for errors. During production a system administrator would do the same. In this way a system administrator will have one place to look for errors for all applications using ARM.

The content, format, and delivery mechanism for error notifications is not part of this standard. It is implementation dependent. A good implementation will provide sufficient detail to not only detect that a problem occurred, but to also isolate and resolve the problem.

8.2 Errors the Application Should Test For

There is one situation in which the application needs to test for errors. This situation is when the application is creating factory classes. When doing so, there are three exceptions, all thrown by the static methods of `java.lang.Class`, that the application should be prepared to catch. They most likely indicate that the ARM environment has not been initialized correctly. For example, an ARM implementation may not be in the class path, or the `java.lang.properties` file may contain invalid class names. These exceptions are:

- `java.lang.ClassNotFoundException` signals that a class to be loaded could not be found. It is thrown by `Class.forName()`.
- `java.lang.IllegalAccessException` signals that a class or initializer is not accessible. It is thrown by `Class.newInstance()`.
- `java.lang.InstantiationException` signals an attempt to instantiate an interface or an abstract class. It is thrown by `Class.newInstance()`.

Chapter 9

The ARM 3.0 Data Models

9.1 Summary

9.1.1 Mandatory Data

ARM requires the following mandatory identification data:

- A 16-byte transaction ID uniquely identifies the type of transaction. Examples are “Query Balance”, “Purchase Product”, and “Backup Data”. The recommended format for the 16-byte ID is the Universally Unique Identifiers (UUID) standard, originally created as part of DCE 1.1.

  The transaction ID is optionally associated to two character strings, the name of the transaction and the name of the application.

ARM requires the following mandatory measurement data. All transaction measurements include these three values.

- The status of a transaction can take one of four values: Good, Failed, Aborted, and Unknown.
- The response time of a transaction, measured in nanoseconds. (Although nanoseconds is overkill for any current application, an 8-byte long value is used, so even when measuring in nanoseconds, this will hold a value of 292 years. The alternative, a 4-byte integer value, would have limited the interface forever to millisecond granularity.)
- The timestamp when the transaction stops, from which, when combined with the response time, the start timestamp can be calculated.

9.1.2 Optional Data

ARM defines the following optional identification data.

- A UUID of a user definition. The user definition contains a character string of a user name. There are no restrictions or implied assumptions on what constitutes a “user name”. Other parameters might be added to the user definition later.
- A unit of work token, called a correlator, that is intended to uniquely identify an instance of a transaction across any JVM on any system. It is used, in conjunction with the parent correlator described next, to show the relationship between transaction instances.

- A unit of work token, called the parent correlator, which indicates the transaction instance that spawned this transaction.
- An 8-byte transaction handle, combined with the transaction ID, uniquely identifies an instance of a transaction within one JVM. Handles were fundamental to the handshaking in ARM 1.0 and ARM 2.0. In ARM 3.0 for Java, they are not needed for handshaking at all. Their purpose is to provide a field in the correlator to differentiate between instances. They can also be useful for problem diagnosis. Applications can ignore the field.
- Zero to seven UUIDs of metric definitions. A metric definition defines the name and format of data values (named "metrics" in ARM), either numeric or string, that represent other interesting data related to a transaction instance. Some examples are the number of record processed, an error code, and a measure of congestion when the transaction is invoked.

ARM defines the following optional measurement data.

- Zero to seven data values, either numeric or string, for the metrics described above.

9.1.3 When to Create a New Identifier (UUID)

A UUID is assigned for three definition types: transaction, user, and metric. This raises an obvious question – when should a UUID be reused versus when should a new UUID be created? The rule is very simple. Each definition contains one or more attributes in addition to the UUID. If any of these attributes change, a new (and by definition unique) UUID should be generated. The application developer should assume that ARM implementations use UUIDs as data keys.

For example, say a metric definition has three attributes, the UUID, the format of the metric, and the name of the metric. Assume the values are UUID=5732 (of course, this would really be a 16-byte value), format=Counter32, name="Bytes Transferred on Connection". This triplet (5732, Counter32, "Bytes Transferred on Connection") is registered with an ARM implementation. Anytime the ARM implementation sees UUID 5732, it knows how to store and report the data.

The application designer decides that a 32-bit counter is not sufficiently large and changes it to a 64-bit counter, and also changes the name to "Bytes Transferred on Session". If the application does not change the UUID, and instead registers the triplet (5732, Counter64, "Bytes Transferred on Session") the results are unpredictable. There would probably be versions using both triplet combinations deployed simultaneously. There would be conflicts in data tables and reports at the least.

The correct procedure is to generate a new UUID, and register a triplet such as (987654, Counter64, "Bytes Transferred on Session"). The original (5732, Counter32, "Bytes Transferred on Connection") triplet remains registered as well. The results are now predictable and there should be no conflicts.

9.2 Data Model Using ArmTransaction

Figure 17 shows the data model when using `ArmTransaction`. To avoid clutter in the diagrams ArmUUID and ArmCorrelator are not shown as separate classes.

If metrics are not used, only ArmTransaction needs to be used. If metrics are used, the ArmTransaction object contains Java references to the ArmMetric objects. Only `ArmTransaction` must be used. All others are optional.

- **ArmTransaction.** `ArmTransaction` is the only interface that must be used. An application typically might create a pool of `ArmTransaction` instances. When a transaction executes the application uses the paired methods `start()` and `stop()` to indicate the beginning and end of the transaction. This is analogous to ARM 1.0/2.0. The measurements for all transaction instances are status, response time, and time of day when stopped. Optionally the application can also provide data to correlate parent and child transactions and the UUID of a user definition. The ARM implementation maintains a unique handle. If metrics are used, substitute ArmTransactionWithMetrics.

- **ArmTransactionWithMetrics**. `ArmTransactionWithMetrics` is a subclass of `ArmTransaction`. If metrics are used (see the following descriptions of `ArmMetric` and `ArmMetricGroup`), `ArmTransactionWithMetrics` is used instead of `ArmTransaction`.

- **ArmMetric (and its ten subclasses, such as ArmMetricCounter32).** The application can optionally augment the basic and correlation data with other data, called metrics. A metric is a numeric or string value. It may represent a counter, such as the number of files processed, a gauge, such as the queue length when the transaction executes, or information such as an error number or the name of a file that was processed. An application creates `ArmMetric` instances and associates them to one or more `ArmTransactionWithMetrics` instances. When `ArmTransactionWithMetrics` methods are processed the values in the `ArmMetric` instances are captured and considered part of the data for the transaction.

  A benefit of this approach is that the same ArmMetric object can be shared by many instances of the same transaction or different transactions. Updating the value in one place (the ArmMetric object) effectively propagates it to many ArmTransactionWithMetrics objects, though the data is only captured when a start(), update(), or stop() call is made.

- **ArmMetricGroup.** `ArmMetricGroup` serves as a notational convenience, grouping one to seven metrics into a set. If an `ArmTransactionWithMetrics` uses metrics, references to each metric are grouped together in an `ArmMetricGroup`, which is passed to the `ArmTransactionWithMetrics` when it is created. The `ArmTransactionWithMetrics` extracts the references from the `ArmMetricGroup` and binds to them directly. After this point in time the `ArmMetricGroup` is not needed any more.

- **ArmUserDefinition, ArmTranDefinition, and ArmMetricDefinition.** The use of these classes is optional. The use was described in the preceding section.

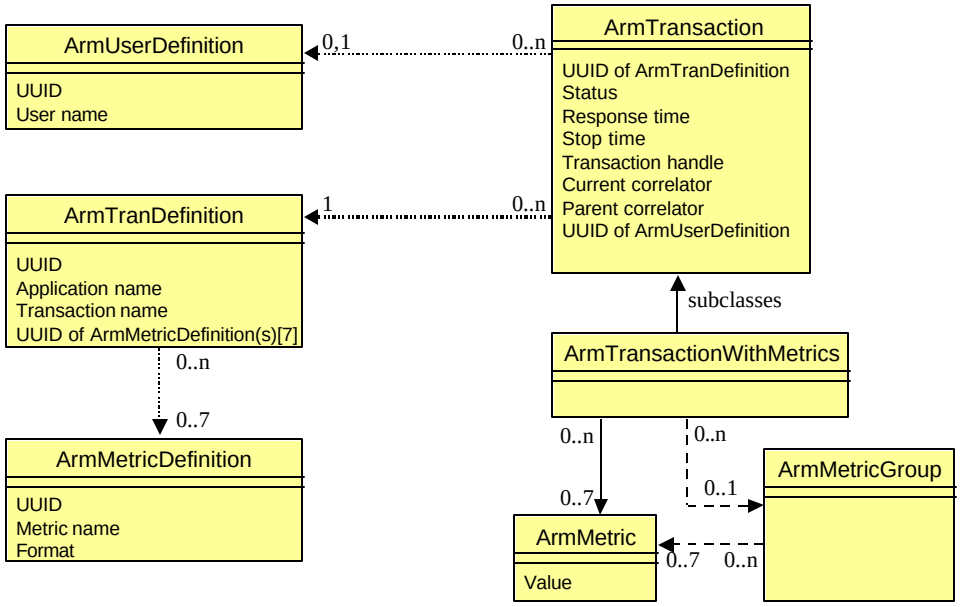

Figure 17. ARM 3.0 Data Model Using ArmTransaction

Logically, all the associations shown are equivalent. `ArmMetric` is an abstract class. The real classes are subclasses, such as `ArmMetricCounter32`. The reason the association from `ArmTransaction` to `ArmMetric` in the second figure is a solid line is that it is a direct reference between Java objects. The dashed lines to/from `ArmMetricGroup` indicate that these are direct Java references, but they are transient, because the information is extracted and stored in `ArmTransaction`. The other associations are shown as dotted lines because it is the value of the UUID attributes that connects them. A further diagrammatic convention is the use of arrows to show that the associations are one-way in each case.

9.3 Data Model Using ArmTranReport

Figure 17 and Figure 18 summarize the data model. The model is based on the DMTF/CIM Distributed Application Performance schema. To avoid clutter in the diagrams ArmUUID and ArmCorrelator are not shown as separate classes.

Figure 18 shows the data model when using `ArmTranReport`. Each `ArmTranReport` instance is a standalone object with no dependencies on any other object. Its association to other objects is indirectly through the UUID of the `ArmTranDefinition` and optionally an `ArmUserDefinition`. The dotted lines in the figures are meant to indicate that this is not a direct Java reference.

- **ArmTranReport, ArmTranReportWithMetrics.** `ArmTranReport` and `ArmTranReportWithMetrics`, its subclass, contain the information about a completed transaction. Typically an application might create one `ArmTranReport` instance for each type of transaction that it executes, or a pool of them if it is multi-threaded. When a transaction completes, the application extracts one of the records, populates it, then calls

`process()`. As soon as `process()` returns the application can reuse the `ArmTranReport` instance.

- **ArmUserDefinition.** `ArmUserDefinition` is provided as a convenience to the application. `ArmUserDefinition` maps a UUID to a user name. Other properties may be added in the future.
- **ArmTranDefinition.** `ArmTranDefinition` is provided as a convenience to the application. It maps a UUID to the application and transaction names, and to zero to seven UUIDs of metric definitions. Other properties may be added in the future.
- **ArmMetricDefinition.** `ArmMetricDefinition` is provided as a convenience to the application. `ArmMetricDefinition` maps a UUID to a name and to the format of a metric. Other properties may be added in the future.

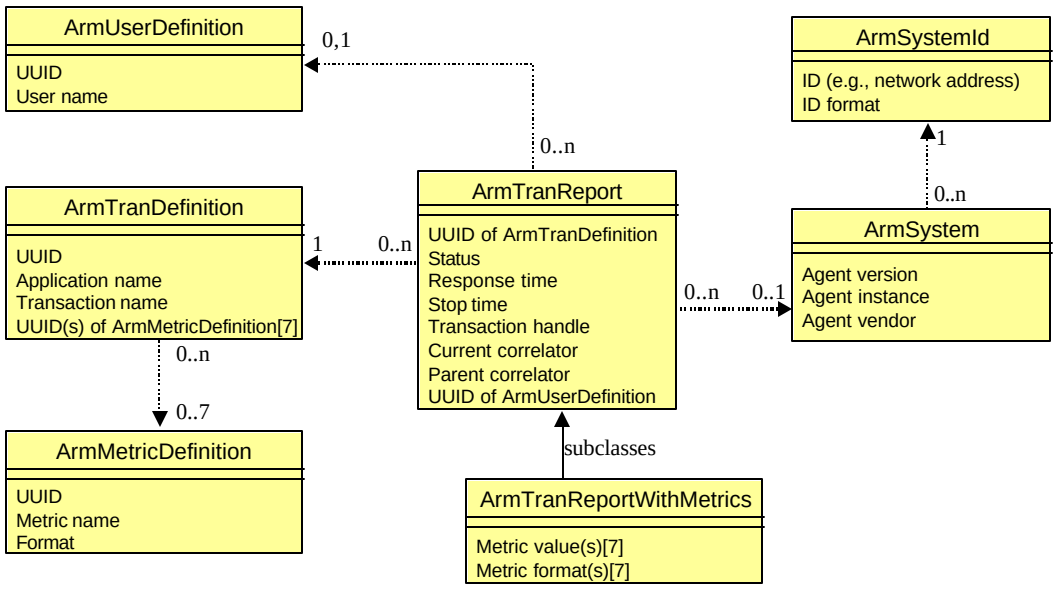

Figure 18. ARM 3.0 Data Model Using ArmTranReport

Chapter 10

The org.opengroup.arm3. Packages*

10.1 Interface list by Java Package

The following table lists all the interfaces, arranged by package name, and by alphabetical order within the package. Most applications will use only the transaction package and can ignore the other packages, which address specialized requirements.

There is one factory interface for each package. Use this factory interface to create instances of the other interfaces.

org.opengroup.arm3. transaction	**org.opengroup.arm3. tranreport**	**org.opengroup.arm3. metric**	**org.opengroup.arm3. definition**
ArmConstants ArmCorrelator ArmToken ArmTransaction ArmTransactionFactory ArmUUID	ArmSystem ArmSystemId ArmTranReport ArmTranReportCorrelator ArmTranReportFactory ArmTranReportWithMetrics	ArmMetric ArmMetricCounter32 ArmMetricCounter64 ArmMetricCounterFloat32 ArmMetricFactory ArmMetricGauge32 ArmMetricGauge64 ArmMetricGaugeFloat32 ArmMetricGroup ArmMetricNumericId32 ArmMetricNumericId64 ArmMetricString32 ArmMetricString8 ArmTransactionWithMetrics	ArmDefinitionFactory ArmMetricDefinition ArmTranDefinition ArmUserDefinition

10.2 Interface List in Alphabetical Order

The following table lists in alphabetical order all the interfaces that comprise the standard, and lists the Java package in which they can be found. To create instances of the class, use the appropriate method of the factory class for the package.

Class Name (alphabetically)	**Java Package**
ArmConstants	org.opengroup.arm3.transaction
ArmCorrelator	org.opengroup.arm3.transaction

Class Name (alphabetically)	Java Package
ArmDefinitionFactory	org.opengroup.arm3.definition
ArmMetric	org.opengroup.arm3.metric
ArmMetricCounter32	org.opengroup.arm3.metric
ArmMetricCounter64	org.opengroup.arm3.metric
ArmMetricCounterFloat32	org.opengroup.arm3.metric
ArmMetricDefinition	org.opengroup.arm3.definition
ArmMetricFactory	org.opengroup.arm3.metric
ArmMetricGauge32	org.opengroup.arm3.metric
ArmMetricGauge64	org.opengroup.arm3.metric
ArmMetricGaugeFloat32	org.opengroup.arm3.metric
ArmMetricGroup	org.opengroup.arm3.metric
ArmMetricNumericId32	org.opengroup.arm3.metric
ArmMetricNumericId64	org.opengroup.arm3.metric
ArmMetricString32	org.opengroup.arm3.metric
ArmMetricString8	org.opengroup.arm3.metric
ArmSystem	org.opengroup.arm3.tranreport
ArmSystemId	org.opengroup.arm3.tranreport
ArmToken	org.opengroup.arm3.transaction
ArmTranDefinition	org.opengroup.arm3.definition
ArmTranReport	org.opengroup.arm3.tranreport
ArmTranReportCorrelator	org.opengroup.arm3.tranreport
ArmTranReportFactory	org.opengroup.arm3.tranreport
ArmTranReportWithMetrics	org.opengroup.arm3.tranreport
ArmTransaction	org.opengroup.arm3.transaction
ArmTransactionFactory	org.opengroup.arm3.transaction
ArmTransactionWithMetrics	org.opengroup.arm3.metric
ArmUserDefinition	org.opengroup.arm3.definition
ArmUUID	org.opengroup.arm3.transaction

10.2.1 Method Naming Conventions

Every attempt has been made to adhere to the naming conventions commonly used in Java programs. Here is a list of a few to be aware of.

- `get()` or `getSomething()` returns a primitive value or a reference to an object or array. If it is a reference, the object (or array) to which it refers is treated as immutable. Treating the object or array as immutable means the ARM implementation will not change the data in the object or array. Some object types are entirely immutable, including `String`, `ArmToken` and its subclasses (`ArmCorrelator`, `ArmSystemId`, `ArmUUID`), and `ArmSystem`. Some object types will not be changed by the ARM implementation, but could be changed by the application (an array or an object that implements `ArmMetric`).
- `set()` or `setSomething()` sets a primitive value or a reference to an object or array.
- `copySomething(destination)` copies the actual data to a byte array. It does not merely copy the reference.
- `isSomething()` returns a boolean.
- `newSomething()` creates a Something object and returns a reference to it.

10.3 org.opengroup.arm3.transaction.ArmConstants

Defines some commonly used constants by several classes in the various packages.

Note that the slots are numbered from one to seven to remain consistent with previous versions of ARM. This is different than the way that Java arrays are indexed. The first element in a Java array is numbered zero, not one.

```
public interface ArmConstants {
// No Public Constructors
// Constants
    public static final int ARM_ABORT;              // Valid status value for ArmTranReport
                                                    // and ArmTransaction (=1)
    public static final int ARM_FAILED;             // Valid status value for ArmTranReport
                                                    // and ArmTransaction (=2)
    public static final int ARM_GOOD;               // Valid status value for ArmTranReport
                                                    // and ArmTransaction (=0)
    public static final int ARM_INVALID;            // Status value used when application
                                                    // passes an invalid value (=-1)
    public static final int ARM_UNKNOWN;            // Valid status value for ArmTranReport
                                                    // and ArmTransaction (=3)
    public static final int CORR_FORMAT_UNKNOWN; // Correlator format for a correlator that was
                                                    // created from erroneous data (=127)
    public static final int CORR_MAX_LENGTH;        // Maximum length of a correlator
                                                    // (currently = 168 bytes)
    public static final int CORR_MIN_LENGTH;        // Minimum length of a correlator
                                                    // (currently = 4 bytes)
    public static final int NAME_MAX_LENGTH;        // Maximum length of a name in
                                                    // characters (currently = 128 characters)
    public static final int SLOTS;                  // Number of metric slots (currently = 7)
    public static final int SLOT_MAX;               // Maximum value of a metric slot (currently = 7)
    public static final int SLOT_MIN;               // Minimum value of a metric slot (currently = 1)
    public static final int UUID_LENGTH;            // Length of all UUIDs (16 bytes)
// No Instance Methods
}
```

10.4 org.opengroup.arm3.transaction.ArmCorrelator

Implements a correlation token passed from a calling transaction to a called transaction. A correlator contains a two-byte length field, a one-byte format ID, a one-byte flag field, plus it may contain other data that is used to uniquely identify an instance of a transaction. All correlators must adhere to the constraints described in Table 1. ARM Correlator Format Constraints on page 90. As long as the correlator byte array is created by a standards-compilant ARM implementation, these constraints will be satisfied. Applications do not need to understand correlator formats.

The use of correlators is described in the "Understanding the Relationships Between Transactions" section of this document (Chapter 4 on page 13).

`equals(Object obj)`, a method inherited from `java.lang.Object`, returns true if the internal data is byte-for-byte identical in two objects. For example, `a.equals(b)` returns true if and only if:

- Both a and b implement `ArmCorrelator`
- The inherited methods `a.getBytes()` and `b.getBytes()` would return byte arrays of identical lengths and contents.

public interface **ArmCorrelator** extends ArmToken {
// No Public Constructors
// Public Instance Methods (in addition to those defined in ArmToken)
// (Implementations should also override equals() and hashCode() from java.lang.Object.)
    public byte **getFlags**();
    public byte **getFormat**();
}

10.5 org.opengroup.arm3.definition.ArmDefinitionFactory

Provides methods to create instances of optional classes for defining metadata about transactions, metrics, and users, plus the UUID-to-name mappings. The `format` parameter of `newArmMetricDefinition` must be one of the constants defined in `ArmMetricDefinition`.

See the discussion in the "Creating ARM Objects" section of this document (Chapter 6 on Page25) for a description of how to use this interface.

If a null value is passed as the value of any of the parameters, an object is returned that contains dummy data for the null parameter. For example, a null ArmUUID parameter might result in creating an object in which the UUID value contains sixteen bytes of zeros or is replaced with a UUID that is used in error situations. Different ARM implementations may handle the situation in different ways, but in all cases, they will return an object that is syntactically correct, that is, any of its methods can be invoked without causing an exception.

public interface **ArmDefinitionFactory** {
// Public Constants
    public static final String **propertyKey**;
// Public Instance Methods
    public ArmMetricDefinition **newArmMetricDefinition**(ArmUUID *uuid*, int *format*, String *name*);
    public ArmTranDefinition **newArmTranDefinition**(ArmUUID *uuid*, String *tranName*, String *applName*);
    public ArmUserDefinition **newArmUserDefinition**(ArmUUID *uuid*, String *name*);
}

10.6 org.opengroup.arm3.metric.ArmMetric

This abstract interface is a superclass for all the metric interfaces. At present there are no public behaviors.

Objects that implement a subclass of `ArmMetric` are used with `ArmTransactionWithMetrics`. They are bound to `ArmTransactionWithMetrics` when the `ArmTransactionWithMetrics` instance is created. Each `ArmMetric` instance can be bound to any number of `ArmTransactionWithMetrics` instances. Setting the value of the `ArmMetric` instance effectively sets the value for all the `ArmTransactionWithMetrics` instances to which it is bound.

public interface **ArmMetric** extends Object {
// No Public Constructors
// No Public Instance Methods
}

10.7 org.opengroup.arm3.metric.ArmMetricCounter32

Implements a 32-bit integer counter. It is the same as ARM 2.0 metric type=1 (ARM_Counter32).

The use of this class is described in the “Additional Data About a Transaction” section of this document (Chapter 5 on page 17).

public interface **ArmMetricCounter32** extends ArmMetric {
// No Public Constructors
// Public Instance Methods
    public int **get**();
    public void **set**(int *value*);
}

10.8 org.opengroup.arm3.metric.ArmMetricCounter64

Implements a 64-bit integer counter. It is the same as ARM 2.0 metric type=2 (ARM_Counter64).

The use of this class is described in the “Additional Data About a Transaction” section of this document (Chapter 5 on page 17).

public interface **ArmMetricCounter64** extends ArmMetric {

// No Public Constructors
// Public Instance Methods
    public long **get**();
    public void **set**(long *value*);
}

10.9 org.opengroup.arm3.metric.ArmMetricCounterFloat32

Implements a 32-bit floating-point counter. It is roughly equivalent to the ARM 2.0 metric type=3 (ARM_CntrDivr32). Instead of providing two integer values that can be divided to produce a floating-point value, which is what was done in ARM 2.0, a floating-point value is provided directly. This was not done in ARM 2.0 because ARM would have to support multiple floating point formats, depending on the programming language and / or machine architecture.

The use of this class is described in the "Additional Data About a Transaction" section of this document (Chapter 5 on page 17).

public interface **ArmMetricCounterFloat32** extends ArmMetric {

// No Public Constructors
// Public Instance Methods
public float **get**();
public void **set**(float *value*);
}

10.10 org.opengroup.arm3.definition.ArmMetricDefinition

Each instance of this class represents a metric definition. A metric definition associates a 16-byte UUID with:

- a data format (one of the ten supported formats)
- a name of up to 128 characters.

The UUID, name, and format are input parameters to the `ArmDefinitionFactory` method `newArmMetricDefinition()`. The format must be one of the constants defined in this class (e.g., `METRIC_COUNTER32`). After the factory creates the metric definition object, the application should call `process()` to register the data.

The use of this class is described in the "Providing Descriptive Information" section of this document (Chapter 7 on page 33).

The observant reader may notice that there are no `set()` methods, and therefore wonder why `process()` should need to be called at all. The reason is that this allows flexibility for the future. If additional properties are added, and it isn't desirable or practical to initialize them in the factory method, `set()` methods for the new properties could be added and invoked prior to invoking `process()`.

```
public interface ArmMetricDefinition {
// No Public Constructors
// Constants (all returned by getFormat()
    public static final int METRIC_COUNTER32;          // Matches ARM 2.0 value (=1)
    public static final int METRIC_COUNTER64;          // Matches ARM 2.0 value (=2)
    public static final int METRIC_COUNTER_FLOAT32;    // Matches ARM 2.0 value (=3)
    public static final int METRIC_GAUGE32;            // Matches ARM 2.0 value (=4)
    public static final int METRIC_GAUGE64;            // Matches ARM 2.0 value (=5)
    public static final int METRIC_GAUGE_FLOAT32;      // Matches ARM 2.0 value (=6)
    public static final int METRIC_NUMERIC_ID32;       // Matches ARM 2.0 value (=7)
    public static final int METRIC_NUMERIC_ID64;       // Matches ARM 2.0 value (=8)
    public static final int METRIC_STRING32;           // Matches ARM 2.0 value (=10)
    public static final int METRIC_STRING8;            // Matches ARM 2.0 value (=9)
// Public Instance Methods
    public int getFormat();
    public ArmUUID getUUID();
    public String getName();
    public void process();
}
```

10.11 org.opengroup.arm3.metric.ArmMetricFactory

Provides methods to create instances of the classes in the org.opengroup.arm3.metric package, except `ArmMetricDefinition`, in the `org.opengroup.arm3.definition` package.

See the discussion in the "Creating ARM Objects" section of this document (Chapter 6 on Page25) for a description of how to use this interface.

If a null value is passed as a parameter to `newArmTransactionWithMetrics()`, an object is returned that contains dummy data for the invalid parameter. For example, if the array at "`byte[] uuidBytes`" is less than 16 bytes long, the UUID value might be padded with zeros or replaced with a UUID that is used in error situations. Different ARM implementations may handle the situation in different ways, but in all cases, they will return an object that is syntactically correct, that is, any of its methods can be invoked without causing an exception.

public interface **ArmMetricFactory** {
// Public Constants
public static final String **propertyKey**;
// Public Instance Methods
public ArmMetricCounter32 **newArmMetricCounter32**();
public ArmMetricCounter64 **newArmMetricCounter64**();
public ArmMetricCounterFloat32 **newArmMetricCounterFloat32**();
public ArmMetricGauge32 **newArmMetricGauge32**();
public ArmMetricGauge64 **newArmMetricGauge64**();
public ArmMetricGaugeFloat32 **newArmMetricGaugeFloat32**();
public ArmMetricGroup **newArmMetricGroup**();
public ArmMetricNumericId32 **newArmMetricNumericId32**();
public ArmMetricNumericId64 **newArmMetricNumericId64**();
public ArmMetricString32 **newArmMetricString32**();
public ArmMetricString8 **newArmMetricString8**();
public ArmTransactionWithMetrics **newArmTransactionWithMetrics**(ArmUUID *tranUUID*, ArmMetricGroup *group*);
}

10.12 org.opengroup.arm3.metric.ArmMetricGauge32

Implements a 32-bit integer gauge. It is the same as ARM 2.0 metric type=4 (ARM_Gauge32).

The use of this class is described in the “Additional Data About a Transaction” section of this document (Chapter 5 on page 17).

public interface **ArmMetricGauge32** extends ArmMetric {

// No Public Constructors
// Public Instance Methods
public int **get**();
public void **set**(int *value*);
}

10.13 org.opengroup.arm3.metric.ArmMetricGauge64

Implements a 64-bit integer gauge. It is the same as ARM 2.0 metric type=5 (ARM_Gauge64).

The use of this class is described in the "Additional Data About a Transaction" section of this document (Chapter 5 on page 17).

public interface **ArmMetricGauge64** extends ArmMetric {
// No Public Constructors
// Public Instance Methods
    public long **get**();
    public void **set**(long *value*);
}

10.14 org.opengroup.arm3.metric.ArmMetricGaugeFloat32

Implements a 32-bit floating-point gauge. It is roughly equivalent to the ARM 2.0 metric type=6 (ARM_GaugeDivr32). Instead of providing two integer values that can be divided to produce a floating-point value, which is what was done in ARM 2.0, a floating-point value is provided directly. This was not done in ARM 2.0 because ARM would have to support multiple floating point formats, depending on the programming language and / or machine architecture.

The use of this class is described in the "Additional Data About a Transaction" section of this document (Chapter 5 on page 17).

public interface **ArmMetricGaugeFloat32** extends ArmMetric {
// No Public Constructors
// Public Instance Methods
    public float **get**();
    public void **set**(float *value*);
}

10.15 org.opengroup.arm3.metric.ArmMetricGroup

`ArmMetricGroup` is used to bind objects that implement a subclass of `ArmMetric` to an `ArmTransactionWithMetrics` instance when an `ArmTransactionWithMetrics` is created. `ArmMetricGroup` is a notational convenience. Using it consolidates all the `ArmMetric` bindings in one parameter passed to the `newArmTransactionWithMetrics()` method of `ArmMetricFactory`. The binding is done when `ArmTransactionWithMetrics` is created for performance reasons (so no object creation and / or binding needs to be done while transactions are being measured). After the `newArmTransactionWithMetrics()` method finishes executing the `ArmMetricGroup` is no longer needed. It can be reused or garbage collected.

`slot` must have a value between `ArmConstants.SLOT_MIN` and `ArmConstants.SLOT_MAX`, inclusive (which are currently equated to 1 and 7, respectively). To remain consistent with ARM 2.0, any `ArmMetric` subclass except `ArmMetricString32` can be assigned to slots 1-6 and only `ArmMetricString32` can be assigned to slot 7.

- `set(slot, ArmMetric)` binds an `ArmMetric` instance to one of the seven slots. `set(slot, null)` is equivalent to `clear(slot)`.
- `clear(slot)` sets the value for the slot to `null`.
- `get(slot)` returns the current value for the slot. This value may be `null`.

The use of this class is described in the “ ” section of this document (Section 0 on page 45).

public interface **ArmMetricGroup** extends Object {
// No Public Constructors
// Public Instance Methods
public void **clear**(int *slot*);
public ArmMetric **get**(int *slot*);
public void **set**(int *slot*, ArmMetric *metric*);
}

10.16 org.opengroup.arm3.metric.ArmMetricNumericId32

Implements a 32-bit integer numeric ID. It is the same as ARM 2.0 metric type=7 (ARM_NumericID32).

The use of this class is described in the “Additional Data About a Transaction” section of this document (Chapter 5 on page 17).

```
public interface ArmMetricNumericId32 extends ArmMetric {
// No Public Constructors
// Public Instance Methods
    public int get();
    public void set(int value);
}
```

10.17 org.opengroup.arm3.metric.ArmMetricNumericId64

Implements a 64-bit integer numeric ID. It is the same as ARM 2.0 metric type=8 (ARM_NumericID64).

The use of this class is described in the “Additional Data About a Transaction” section of this document (Chapter 5 on page 17).

public interface **ArmMetricNumericId64** extends ArmMetric {

// No Public Constructors
// Public Instance Methods
public long **get**();
public void **set**(long *value*);
}

10.18 org.opengroup.arm3.metric.ArmMetricString32

Implements a `String` of 1 to 32 characters. It is similar to the ARM 2.0 metric type=10 (ARM_String32), with two differences.

- The characters are in the Java standard UCS-2 format, whereas ARM 2.0 characters are in UTF-8 format.
- The limit of 32 in ARM 2.0 is a byte limit (a character in UTF-8 is represented as 1, 2, or 3 bytes). The limit in the ARM 3.0 Java bindings is a character limit. UCS-2 characters are two bytes in length, so a string of 32 characters will be 64 bytes long.

The use of this class is described in the "Additional Data About a Transaction" section of this document (Chapter 5 on page 17).

public interface **ArmMetricString32** extends ArmMetric {

// No Public Constructors
// Public Instance Methods
    public String **get**();
    public void **set**(String *s*);
}

10.19 org.opengroup.arm3.metric.ArmMetricString8

Implements a `String` of 1 to 8 characters. It is similar to the ARM 2.0 metric type=9 (ARM_String8), with two differences.

- The characters are in the Java standard UCS-2 format, whereas ARM 2.0 characters are in UTF-8 format.
- The limit of 8 in ARM 2.0 is a byte limit (a character in UTF-8 is represented as 1, 2, or 3 bytes). The limit in the ARM 3.0 Java bindings is a character limit. UCS-2 characters are two bytes in length, so a string of 8 characters will be 16 bytes long.

The use of this class is described in the "Additional Data About a Transaction" section of this document (Chapter 5 on page 17).

public interface **ArmMetricString8** extends ArmMetric {

// No Public Constructors
// Public Instance Methods
public String **get**();
public void **set**(String *s*);
}

10.20 org.opengroup.arm3.tranreport.ArmSystem

`ArmSystem` is used when `ArmTranReport` is used to report data about transactions that executed on a different system. In this case the `ArmSystem` should contain the fields of the system on which the transaction executed. When transactions are reported on the same system on which they are measured, the application does not have to use `ArmSystem`. The ARM implementation will provide all the fields (probably as part of its initialization setup, possibly with the input of a system administrator).

Within one JVM a transaction instance is distinguished from all other instances by the combination of the transaction UUID and the transaction handle (a 64-bit integer). Although there is a theoretical potential for duplicate handles, it should be easy for an ARM implementation to prevent them occurring during the lifetime of interest of a transaction record. By themselves the transaction UUID and handle will not prevent conflicts between multiple JVMs on the same system, or between JVMs on different systems. Because correlating transactions needs to work across systems, more fields are needed to disambiguate all the transaction instances and records.

Five fields in `ArmSystem` are provided for this purpose. The combination of all five plus the transaction UUID and handle greatly lessens the potential for collisions.

- **System ID**. The system ID is the network name or address of the system, as it would be sent in a data frame across a network in network byte order. This and the System ID format field are encapsulated in an ArmSystemId object.
- **System ID format**. The format of the system ID, such as an SNA address or a hostname. This and the System ID field are encapsulated in an ArmSystemId object.
- **Instance**. There could be several JVMs running on the same system, all sharing the same system ID, especially if the system ID does not contain a distinguishing identifier such as a port number. This 2-byte field can be set by an administrator to a unique value for each JVM to disambiguate between them.
- **Vendor ID**. Different organizations will provide ARM implementations, and each ARM implementation may have unique capabilities. It could be useful for an analysis program to know that a correlator was generated by an ARM implementation provided by a certain vendor, and at a certain version level. This could help the analysis program know where to look for additional information, for example.

  In order to minimize the possibility of two vendors using the same vendor ID, the value should be taken from the list of enterprise identifiers from the Internet Assigned Numbers Authority (IANA). This list was created for vendors who have SNMP agents. Although the ARM specification does not require or endorse SNMP, it's likely that many of the organizations that will create an ARM agent will have at least one enterprise ID assigned. The list of enterprise IDs can be found at:

  ftp://ftp.isi.edu/in-notes/iana/assignments/enterprise-numbers.

  For organizations that don't have an enterprise identifier assigned by the IANA, the values between 32768-65535 are reserved for agent developers to use. There are no semantics associated with these ids. It is expected that most agent developers will have a formally

assigned vendor id, and it will be an unusual situation where another id is needed, but this provides a solution. There is a risk that two different agent developers will choose the same id, but this risk is deemed acceptably small.

- **Agent Version**. The Agent Version is used to distinguish between different versions of an agent, and will be most useful when the capabilities and / or interfaces of an agent change from one release to another. It will also be useful to distinguish between different agents from the same vendor. Each vendor is responsible for avoiding having multiple agents with different capabilities using the same Vendor ID + Agent Version values.

The fields are set using the `newArmSystem()` method of `ArmTranReportFactory` or the `getArmSystem()` method of `ArmTranReportCorrelator`. There are no `set()` methods for the individual fields. The object is immutable.

public interface **ArmSystem** {
// No Public Constructors
// Public Instance Methods
    public short **getAgentVersion**();
    public short **getInstance**();
    public ArmSystemId **getSystemId**();
    public short **getVendorId**();
}

10.21 org.opengroup.arm3.tranreport.ArmSystemId

Encapsulates the network addressing information for a system. It is used within correlation tokens and as input to ArmSystem.

- **System ID**. The system ID is the network name or address of the system, as it would be sent in a data frame across a network in network byte order, or a UUID that can be mapped to the network name or address.
- **System ID format**. The format of the system ID, such as an SNA address or a hostname.

`equals(Object obj)`, a method inherited from `java.lang.Object`, returns true if the internal data is byte-for-byte identical in two objects. For example, `a.equals(b)` returns true if and only if:

- Both a and b implement `ArmSystemId`
- The inherited methods `a.getBytes()` and `b.getBytes()` would return byte arrays of identical lengths and contents
- `a.getFormat()` and `b.getFormat()` would return identical values.

The fields are set using the `newArmSystemId()` method of `ArmTranReportFactory` or the `getArmSystemId()` method of `ArmSystem`. There are no `set()` methods for the individual fields. The object is immutable.

public interface **ArmSystemId** extends ArmToken {
// public constants
    public static final short **FORMAT_HOSTNAME**;
    public static final short **FORMAT_IPV4**;
    public static final short **FORMAT_IPV4PORT**;
    public static final short **FORMAT_IPV6**;
    public static final short **FORMAT_IPV6PORT**;
    public static final short **FORMAT_SNA**;
    public static final short **FORMAT_UUID**;
    public static final short **FORMAT_X25**;
// No Public Constructors
// Public Instance Methods (in addition to those defined by ArmToken)
// (Implementations should also override equals() and hashCode() from java.lang.Object.)
    public short **getFormat**();
}

10.22 org.opengroup.arm3.transaction.ArmToken

This abstract interface is a superclass for other classes that consist of a byte array data token, plus optionally other data. Subclasses include `ArmCorrelator`, `ArmSystemId`, and `ArmUUID`.

This abstract interface is a superclass of `ArmCorrelator`, `ArmSystemId`, and `ArmUUID`, expressing the common part of their interfaces. ArmToken is abstract in the sense that any ArmToken object returned by any method in this specification satisfies one of the subclass interfaces. Objects of these identify particular entities. These objects contain a byte array data token, plus optionally other identifying data, which together comprise the value of the object.

To make it possible to compare the values of these tokens, and to use these tokens as hashkeys (so that the user can associate data with a particular token), this specification requires that any class implementing any of these types override the `java.lang.Object` methods `equals(Object)` and `hashCode()`. The behavior of these methods must be the following.

`a.equals(b)` returns true if the `ArmToken` objects a and b have the same value (internal data is byte-for-byte identical in two objects). That is, `a.equals(b)` is true if and only if:

- a and b implement the same interface `ArmCorrelator`, `ArmSystemId`, or `ArmUUID`.
- `a.getBytes()` and `b.getBytes()` would return byte arrays of identical lengths and contents.
- If a subclass of `ArmToken` defines other data values (specifically, `ArmSystemId` defines a short field named format), these other data values are all identical.

If `a.equals(b)==true`, `a.hashCode()` and `b.hashCode()` will return the same value. The `hashCode()` value is implementation-dependent. In other words, hashcode values are not necessarily portable. A hashcode generated on one system by one implementation may not equal a hashcode value generated on another system by a different implementation, even if `a.equals(b)==true`.

`copyBytes(byte[] dest)` copies the token's byte array into the destination byte array, which must have a length greater than or equal to the token's byte array.

`copyBytes(byte[] dest, int offset)` copies the token's byte array into the destination byte array at the specified offset. The destination array must be large enough to hold the byte array value, that is, `dest.length-offset >= token.getLength()`.

`getBytes()` returns a newly allocated byte array initialized to the value of the token's byte array.

`getLength()` returns the size of the byte array part of the value.

public abstract interface **ArmToken** extends Object {
// No Public Constructors
// Public Instance Methods
// (Subclasses should also override equals() and hashCode() from java.lang.Object.)
public boolean **copyBytes**(byte[] *dest*);
public boolean **copyBytes**(byte[] *dest*, int *offset*);
public byte[] **getBytes**();
public int **getLength**();

}

10.23 org.opengroup.arm3.definition.ArmTranDefinition

Each instance of this class represents a transaction definition. A transaction definition associates a 16-byte universally unique ID with:

- an application name of up to 128 characters (ApplName)
- a transaction name of up to 128 characters (TranName)
- zero to seven metric definition IDs (also 16-byte universally unique IDs).

The UUID and names are input parameters to the `ArmDefinitionFactory` method `newArmTranDefinition()`. After the factory creates the metric definition object, the application may use `setMetricUUID()` to assign metric UUIDs to any of the seven slots. After setting any metric definitions, the application should call `process()` to register the data.

The use of this class is described in the "Providing Descriptive Information" section of this document (Chapter 7 on page 33).

After `process()` is called, the object should be treated as being immutable. ARM implementations may enforce this by doing nothing when `setMetricUUID()` is executed after `process()` has been executed. This is because the mapping of the UUID to the names and metric UUIDs needs to be unique, as described in Section 9.1.3. Any change to any value requires a new UUID, which can only be set with `newArmTranDefinition()`.

public interface **ArmTranDefinition** {
// No Public Constructors
// Public Instance Methods
public String **getApplName**();
public ArmUUID **getMetricUUID**(int *slot*);
public ArmUUID **getUUID**();
public String **getTranName**();
public void **process**();
public void **setMetricUUID**(int *slot*, ArmUUID *uuid*);
}

10.24 org.opengroup.arm3.tranreport.ArmTranReport

Applications use `ArmTranReport` to report the results of a transaction that completed previously. The application would have measured the response time. The transaction could have executed on the local system or on a remote system. If it executes on a remote system, `ArmSystem` is used to provide the addressing information for the remote system and the JVM instance on it. If the application uses metrics, the subclass `ArmTranReportWithMetrics` should be used. It has all the behaviors of `ArmTranReport` plus the metric behaviors.

The fundamental rule for correct usage of `ArmTranReport` is to execute a pair of methods for each transaction instance, one from the "init" family, `init()` or `initGenCorr()`, and one of the `process()` methods. The reason this rule must be followed is that the handle, which distinguishes transaction instances from each other, and all other identity information, is set when one of the "init" methods is executed. Executing two `process()` methods in a row will result in a duplicate transaction handle, and conflicting data when it is processed, such as in a reporting application.

The "init" methods establish the identity of a transaction. The identity includes the transaction UUID, the transaction handle, the parent correlator (if any), and the five fields in `ArmSystem`.

- `init()` requires the transaction UUID. Optionally the `ArmSystem` and/or parent correlator can be provided. If no `ArmSystem` is provided, the ARM implementation uses the default for the local system. `init()` generates a new transaction handle and stores it internal to the `ArmTranReport` object.

  Because an `ArmTranReportCorrelator` contains the transaction UUID, `ArmSystem`, and transaction handle, it can be used as an input parameter instead of the transaction UUID and `ArmSystem`. In this case, a new transaction handle is not generated. The handle in the `ArmTranReportCorrelator` is used.

- `initGenCorr()` is identical to `init()` except that it also generates a correlator and returns a reference to it. The application can use the `ArmToken` methods `copyBytes()` to copy its data into an existing byte array, or the `getBytes()` method to create a new array, populated with the data. The program needs the byte array values if it is going to send it to a partner program, like a server, so the transactions can be linked together.

The `process()` method reports the measurement data, and a user UUID if there is one. It is executed once per measured transaction. It should not be executed again until one of the "init" family of methods is executed. The `status` value of must be one of `ARM_ABORT`, `ARM_FAILED`, `ARM_GOOD`, or `ARM_UNKNOWN` (all defined in `ArmConstants`). The response time is the time in nanoseconds. The stop time format is the same as the Java system clock returned from `java.lang.System.currentTimeMillis()`.

To correlate transactions using `ArmTranReport`, if there are child transactions, `initGenCorr()` is executed **prior** to the transaction being executed. This is necessary because the correlator is passed to any child transactions, and therefore it must be available before the child transactions are invoked. **`process()`** is executed after the execution completes, when the status and response time are known.

The `get()` methods can be used to see the data within an object. Ordinarily an application would have no need to use them. `getCorr()`, `getParentCorr()`, `getTranUUID()`, `getUserUUID()`, and `initGenCorr()` will return immutable objects.

`getTranHandle()` will return a positive number. Otherwise there are no specified semantics on how one is generated, as long as it is unique for a combination of the transaction UUID plus the `ArmSystem` parameters that are in the correlator (which the application need not be concerned with).

```
public interface ArmTranReport {
// No Public Constructors
// Public Instance Methods
    public ArmTranReportCorrelator getCorr();
    public ArmCorrelator getParentCorr();
    public long getResponseTime();
    public int getStatus();
    public long getStopTime();
    public long getTranHandle();
    public ArmUUID getTranUUID();
    public ArmUUID getUserUUID();
    public void init(ArmUUID tranUUID);
    public void init(ArmUUID tranUUID, ArmSystem sys);
    public void init(ArmUUID tranUUID, ArmSystem sys, ArmCorrelator parentCorr);
    public void init(ArmUUID tranUUID, ArmCorrelator parentCorr);
    public void init(ArmTranReportCorrelator corr);          // Uses uuid, handle, and
                                                             // ArmSystem values in corr
    public void init(ArmTranReportCorrelator corr, ArmCorrelator parentCorr);
                                                             // Uses uuid, handle, and ArmSystem
                                                             // values in corr
    public ArmTranReportCorrelator initGenCorr(ArmUUID tranUUID);
    public ArmTranReportCorrelator initGenCorr(ArmUUID tranUUID, ArmSystem sys);
    public ArmTranReportCorrelator initGenCorr(ArmUUID tranUUID, ArmSystem sys, ArmCorrelator
    parentCorr);
    public ArmTranReportCorrelator initGenCorr(ArmUUID tranUUID, ArmCorrelator parentCorr);
    public void process(int status, long respTime);
    public void process(int status, long respTime, long stopTime);
    public void process(int status, long respTime, long stopTime, ArmUUID userUUID);
}
```

10.25 org.opengroup.arm3.tranreport.ArmTranReportCorrelator

Represents a particular correlator format. A correlator of this format contains the following fields. The ARM Format=2 correlator (described in Table 3. ARM Correlator Format 2 on page 90) is an example of a correlator format with these fields. The methods are accessor methods that return these values.

- A sixteen-byte UUID representing the transaction UUID. It is the same parameter as the `tranUUID` parameter passed to the `init()` method of `ArmTranReport`.
- An eight-byte long representing the transaction handle. It is the same value returned by the `getTranHandle()` method of `ArmTranReport`.
- The parameters contained within `ArmSystem`.

If using `ArmTranReportCorrelator`, care must be taken that the correlator byte array passed to the `ArmTranReportFactory` method `newArmTranReportCorrelator()` is a correlator that contains this data. The surest way to insure this is to provide a correlator in format 2, described in Table 3. ARM Correlator Format 2 on page 90. If the ARM implementation does not recognize the data it is passed, `newArmTranReportCorrelator()` will return an object that contains dummy data (such as all zeros for most of the data fields.) This object is syntactically correct, that is, any of its methods can be invoked without causing an exception, though the data in the correlator may be meaningless and may not be unique. This is more than a hypothetical distinction, because the correlator format 1 defined by the ARM 2.0 specification (see Table 2. ARM Correlator Format 1 on page 90) is not of this format. Using `ArmTransaction` avoids this complication.

As a subclass of `ArmCorrelator`, all the behaviors of `ArmCorrelator` apply.

```
public interface ArmTranReportCorrelator extends ArmCorrelator {
// No Public Constructors
// Public Instance Methods
    public ArmSystem getArmSystem();
    public long getTranHandle();
    public ArmUUID getTranUUID();
}
```

10.26 org.opengroup.arm3.tranreport.ArmTranReportFactory

Provides methods to create instances of the classes in the `org.opengroup.arm3.tranreport` package.

See the discussion in the "Creating ARM Objects" section of this document (Chapter 6 on Page25) for a description of how to use this interface.

If a null value is passed as the value of any of the parameters, or the parameters are otherwise invalid (such as an offset that extends beyond the end of an array), an object is returned that contains dummy data for the invalid parameter. For example, a null "`byte[] corrBytes`" parameter might result in creating an object with a correlator that contains mostly zeros. Different ARM implementations may handle the situation in different ways, but in all cases, they will return an object that is syntactically correct, that is, any of its methods can be invoked without causing an exception.

public interface **ArmTranReportFactory** {
// Public Constants
    public static final String **propertyKey**;
// Public Instance Methods
    public ArmSystem **newArmSystem**(); // Uses the default values for the local system
    public ArmSystem **newArmSystem**(ArmSystemId sys*id*, short *instance*, short *vendorId*, short *agentVer*);
    public ArmSystemId **newArmSystemId**(short *format*, byte[] *idBytes*);
    public ArmSystemId **newArmSystemId**(short *format*, byte[] *idBytes*, int *offset*);
    public ArmSystemId **newArmSystemId**(short *format*, byte[] *idBytes*, int *offset*, int *length*);
    public ArmTranReport **newArmTranReport**();
    public ArmTranReportCorrelator **newArmTranReportCorrelator**(byte[] *corrBytes*);
    public ArmTranReportCorrelator **newArmTranReportCorrelator** (byte[] *corrBytes*, int *offset*);
    public ArmTranReportWithMetrics **newArmTranReportWithMetrics**();
}

10.27 org.opengroup.arm3.tranreport.ArmTranReportWithMetrics

`ArmTranReport` does not support the use of metrics. `ArmTranReportWithMetrics` is a subclass of `ArmTranReport` that adds the metric support.

The use of metrics is described in the "Additional Data About a Transaction" section of this document (Chapter 5 on page 17).

The metrics are used in addition to `ArmTranReport` semantics. The values are set or cleared prior to executing the `ArmTranReport process()` method. They can be executed before or after one of the "init" family of methods is executed. Note that unlike `ArmTransactionWithMetrics`, `ArmTranReportWithMetrics` does not use separate `ArmMetric` objects to contain the metric values. The values are set directly into the `ArmTranReportWithMetrics` object.

Metrics are assigned to one of seven slots, which are numbered from 1 to 7. In a Java program, these fixed indexes (the slots) are rather clumsy. This isn't the best way to do it. However, backwards compatibility with previous versions of ARM requires them. The analysis and reporting programs that support ARM metrics would require major overhauls if a different data model is used, and this is unacceptable. ARM 2.0 defined a C language interface, and metrics were provided at fixed offsets into a buffer, and the metric definitions were done the same way. This explains why some aspects of the metric interface look and feel like a procedural interface, rather than an object-oriented interface.

An application using metrics needs to take care to use the same slot indexes that are used for the definitions. If the metric definition states that there is a 64-bit gauge in slot 5, the application is obliged to only store 64-bit gauge data in that slot using `setMetricGauge64(5, value)`, for example. Storing different data should not cause a runtime error but it will cause an error when the data are reported.

There are 23 methods, but it's easier to think of them as five method types.

- `clearMetric(slot)` indicates that the data in the slot has no meaning at this time.
- `getMetricType(slot)` returns one of the ten constants defined in `ArmMetricDefinition`, which indicates the format of the data in the slot.
- `isMetricSet(slot)` indicates if the data in the slot has meaning at this time.
- The ten `getMetricXYZ(slot)` methods, where "XYZ" is replaced with one of the ten metric types, returns the current value, whether it is valid or not. If the format part of the method name (e.g., "Counter32") does not match the format part used to set the data, the returned data is unpredictable. For example, if `setMetricString8(1,"Update")` set a string value in slot 1, and `getMetricCounter32(1)` requests an integer value, the returned data value is unpredictable. Such a request, even though it is invalid, will not cause a runtime error or an exception to be thrown. Similarly, unpredictable data will be returned, and no exception thrown, if `isMetricSet(slot)==false` for the specified slot.
- The ten `setMetricXYZ(slot, value)` methods, where "XYZ" is replaced with one of the ten metric types, sets the current value and marks the data as valid.

public interface **ArmTranReportWithMetrics** extends ArmTranReport {

```
// No Public Constructors
// Public Instance Methods
    public void clearMetric(int slot);
    public int getMetricCounter32(int slot) ;
    public long getMetricCounter64(int slot) ;
    public float getMetricCounterFloat32(int slot) ;
    public int getMetricGauge32(int slot) ;
    public long getMetricGauge64(int slot) ;
    public float getMetricGaugeFloat32(int slot) ;
    public int getMetricNumericId32(int slot) ;
    public long getMetricNumericId64(int slot) ;
    public String getMetricString32(int slot) ;
    public String getMetricString8(int slot);
    public int getMetricType(int slot);
    public boolean isMetricSet(int slot);
    public void setMetricCounter32(int slot, int value);
    public void setMetricCounter64(int slot, long value);
    public void setMetricCounterFloat32(int slot, float value);
    public void setMetricGauge32(int slot, int value);
    public void setMetricGauge64(int slot, long value);
    public void setMetricGaugeFloat32(int slot, float value);
    public void setMetricNumericId32(int slot, int value);
    public void setMetricNumericId64(int slot, long value);
    public void setMetricString32(int slot, String value);
    public void setMetricString8(int slot, String value);
}
```

10.28 org.opengroup.arm3.transaction.ArmTransaction

For most applications, `ArmTransaction` is the most important of all the ARM classes, and the most widely used. Instances of `ArmTransaction` represent transactions when they execute. The application creates as many instances as it needs. This will typically be at least as many as the number of transactions that can be executing simultaneously. An application may create a pool of `ArmTransaction` objects, take one from the pool to use when a transaction starts, and put it back in the pool after the transaction ends for later reuse. Another strategy is to create one per thread.

`start()`, `update()`, and `stop()` are used to indicate that transactions are starting, continuing to execute, or ending, respectively. The `status` value of `stop()` must be one of `ARM_ABORT`, `ARM_FAILED`, `ARM_GOOD`, or `ARM_UNKNOWN` (all defined in `ArmConstants`).

After `start()` executes the object should not be reused for a different transaction instance until either `stop()` or `reset()` are used. As soon as `stop()`, or `reset()` complete, the object can be reused. There is no need to save any of the internal state from the previous transaction. The state will have already been captured. If two `start()` methods are executed without an intervening `stop()`, the first `start()` is ignored. The ARM implementation may also report the error to a system administrator in some out-of-band way that is implementation-specific, but it will not report the error back to the application.

There are four versions of `start()`, depending on whether a user UUID and / or a parent correlator is provided.

Each instance of a transaction needs to have a unique identifier in case a correlator is generated for it. The ARM specification defines internal formats that should make the correlator unique. Applications need not be concerned with the contents of a correlator.

There are several optional features.

- `getCorr()` returns a reference to the correlator for the current transaction. It will return a newly created immutable object. It can be executed anytime after `start()` is executed until the next `start()` or `reset()` is executed, and it will return an object with the same value each time.
- If metrics are provided, the subclass `ArmTransactionWithMetrics`, in the `org.opengroup.arm3.metrics` package, should be used. See its description for details.
- If no transaction is currently executing, i.e., there has been no `start()` since the last `stop()`, `getResponseTime()` returns the response time of the last completed transaction. If a transaction is currently executing, i.e., there has been no `stop()` since the last `start()`, `getResponseTime()` returns the elapsed time since the current transaction started.

Except for `getCorr()`, applications will normally not use any of the `get()` methods. `getCorr()`, `getParentCorr()`, `getTranUUID()`, and `getUserUUID()` will return newly created immutable objects.

`getTranHandle()` will return a positive number. Otherwise there are no specified semantics on how one is generated, as long as it is unique for a combination of the transaction UUID plus the `ArmSystem` parameters that are in the correlator (which the application need not be concerned with).

```
public interface ArmTransaction {
// No Public Constructors
// Public Instance Methods
    public ArmCorrelator getCorr();
    public ArmCorrelator getParentCorr();
    public long getRespTime();
    public int getStatus();
    public long getStopTime();
    public long getTranHandle();
    public ArmUUID getTranUUID();
    public ArmUUID getUserUUID();
    public void reset();
    public void start();
    public void start(ArmCorrelator parentCorr);
    public void start(ArmUUID userUUID);
    public void start(ArmUUID userUUID, ArmCorrelator parentCorr);
    public long stop(int status);
    public synchronized void update();
}
```

10.29 org.opengroup.arm3.transaction.ArmTransactionFactory

This class is used to create instances of the classes in the `org.opengroup.arm3.transaction` package.

See the discussion in the "Creating ARM Objects" section of this document (Chapter 6 on Page25) for a description of how to use this interface.

No length field is passed to newArmUUID() because all UUIDs are the same length (16 bytes).

No length field is passed to newArmCorrelator() because ARM requires that the length of the correlator be found in the first two bytes of the byte array (either at corrBytes or corrBytes+offset). For more details, see Table 1. ARM Correlator Format Constraints on page 90.

If a null value is passed as the value of any of the parameters, or the parameters are otherwise invalid (such as an offset that extends beyond the end of an array), an object is returned that contains dummy data for the invalid parameter. For example, a null "`byte[] corrBytes`" parameter might result in creating an object with a correlator in format 127 (see Table 4. ARM Correlator Format 127 on page 90.) In another example, if the array at "`byte[] uuidBytes`" is less than 16 bytes long, the UUID value might be padded with zeros or replaced with a UUID that is used in error situations. Different ARM implementations may handle the situation in different ways, but in all cases, they will return an object that is syntactically correct, that is, any of its methods can be invoked without causing an exception, even if the data may be at least partially meaningless.

public interface **ArmTransactionFactory** {
// Public Constants
    public static final String **propertyKey**;
// Public Instance Methods
    public ArmCorrelator **newArmCorrelator**(byte[] *corrBytes*);
    public ArmCorrelator **newArmCorrelator**(byte[] *corrBytes*, int *offset*);
    public ArmTransaction **newArmTransaction**(ArmUUID *tranUUID*);
    public ArmUUID **newArmUUID** (byte[] *uuidBytes*);
    public ArmUUID **newArmUUID** (byte[] *uuidBytes*, int *offset*);
}

10.30 org.opengroup.arm3.metric.ArmTransactionWithMetrics

This subclass of `ArmTransaction` is used if the application wishes to use metrics. All the `ArmTransaction` rules for using `start()`, `stop()`, etc., apply to `ArmTransactionWithMetrics`. It extends `ArmTransaction` by adding methods to manipulate metrics.

- If metrics are provided the `ArmMetric` subclass objects are bound to an `ArmTransactionWithMetrics` object when the `ArmTransactionWithMetrics` is created. This is done by specifying `ArmMetricGroup` in the `newArmTransactionWithMetrics()` method of `ArmMetricFactory`. The `ArmMetricGroup` instance is not used after the `ArmTransactionWithMetrics` instance has been created.
- The value that `getMetric(slot)` returns depends on the value in `ArmMetricGroup` for that slot when `ArmTransactionWithMetrics` is created. If the `get(slot)` method of `ArmMetricGroup` would have returned a reference to an object that implements a subclass of `ArmMetric`, this reference value is returned. If `get(slot)` would have returned `null`, `null` is returned.
- `setMetricValid(slot)` is used to indicate if an ArmMetric subclass assigned to a slot is valid when any of a `start()`, `update()`, or `stop()` call is made. If the valid flag is set then the metric values are processed.
- `isMetricValid(slot)` indicates if an ArmMetric subclass assigned to a slot is valid.

public interface **ArmTransactionWithMetrics** extends ArmTransaction {
// No Public Constructors
// Public Instance Methods
public ArmMetric **getMetric**(int *slot*);
public boolean **isMetricValid**(int *slot*);
public void **setMetricValid**(int *slot*, boolean *value*);
}

10.31 org.opengroup.arm3.definition.ArmUserDefinition

Each instance of this class represents a user definition. A user definition associates a 16-byte universally unique ID with:

- An user name of up to 128 characters

The UUID and names are input parameters to the `ArmDefinitionFactory` method `newArmUserDefinition()`. After the factory creates the metric definition object, the application should call `process()` to register the data.

The use of this class is described in the "Providing Descriptive Information" section of this document (Chapter 7 on page 33).

The observant reader may notice that there are no `set()` methods, and therefore wonder why `process()` should need to be called at all. The reason is that this allows flexibility for the future. If additional properties are added, and it isn't desirable or practical to initialize them in the factory method, `set()` methods for the new properties could be added and invoked prior to invoking `process()`.

```
public interface ArmUserDefinition {
// No Public Constructors
// Public Instance Methods
    public ArmUUID getUUID();
    public String getName();
    public void process();
}
```

10.32 org.opengroup.arm3.transaction.ArmUUID

Implements a 16-byte UUID. UUIDs are used to identify transaction class definitions, user definitions, and metric definitions.

`equals(Object obj)`, a method inherited from `java.lang.Object`, returns true if the internal data is byte-for-byte identical in two objects. For example, `a.equals(b)` returns true if and only if:

- Both a and b implement `ArmUUID`
- The inherited methods `a.getBytes()` and `b.getBytes()` would return byte arrays of identical lengths and contents

public interface **ArmUUID** extends ArmToken {
// No Public Constructors
// No Public Instance Methods (see ArmToken for applicable methods)
// (Implementations should override equals() and hashCode() from java.lang.Object.)
}

Appendix A

Application Instrumentation Sample

```
package org.opengroup.arm3.samples.test;

import java.lang.Math;
import java.net.*;
import java.util.*;
import org.opengroup.arm3.transaction.*;

//--------------------------------------------------------------------
//
// ProducerTranCorrUser
//
// This is a dummy program that does no actual work. To simulate a
// transaction executing, its thread sleeps for a randomly generated
// period of time.
//
//--------------------------------------------------------------------
public class ProducerTranCorrUser
       implements Runnable
{
    private int    numTrans;
    private long   msecBtwTrans;
    private long   sleepTime;
    private Random random;

    private final byte[] uuidTranClientBytes
                           = {1,1,1,1,1,1,1,1,11,11,11,11,11,11,11,11};
    private final byte[] uuidTranAppServerBytes
                           = {2,2,2,2,2,2,2,2,22,22,22,22,22,22,22,22};
    private final byte[] uuidTranDataServerBytes
                           = {3,3,3,3,3,3,3,3,33,33,33,33,33,33,33,33};
    private final byte[] uuidUserBytes
                           = {0,2,4,6,8,10,12,14,16,18,20,22,24,26,28,30};
    private ArmUUID uuidTranClient,
                    uuidTranAppServer,
                    uuidTranDataServer,
                    uuidUser;

    private ArmTransactionFactory   tranFactory;
    private ArmTransaction          tranClient, tranAppServer, tranDataServer;

   //--------------------------------------------------------------------
   // Constructor
   //--------------------------------------------------------------------
```

```
   public ProducerTranCorrUser()
   {
   }

   //---------------------------------------------------------------------
   // init()
   //---------------------------------------------------------------------
   public void init (int numTrans, long msecBtwTrans)
          throws Exception
   {
      this.numTrans     = numTrans;
      this.msecBtwTrans = msecBtwTrans;

      Properties p = System.getProperties();
      String keyTranFactoryClass = ArmTransactionFactory.propertyKey;
      String tranFactoryName     = p.getProperty(keyTranFactoryClass);
      Class tranFactoryClass;
      try
      {  tranFactoryClass = Class.forName(tranFactoryName);
      }
      catch (ClassNotFoundException e)
      {  throw new ClassNotFoundException("Can't find class '" + tranFactoryName +
"'");
      }
      try
      {  tranFactory = (ArmTransactionFactory) tranFactoryClass.newInstance();
      }
      catch (IllegalAccessException e1)
      {  throw new IllegalAccessException("creating instance of '" +
                                          tranFactoryName + "'");
      }
      catch (InstantiationException e2)
      {  throw new InstantiationException("creating instance of '" +
                                          tranFactoryName + "'");
      }

      uuidTranClient     = tranFactory.newArmUUID(uuidTranClientBytes);
      uuidTranAppServer  = tranFactory.newArmUUID(uuidTranAppServerBytes);
      uuidTranDataServer = tranFactory.newArmUUID(uuidTranDataServerBytes);
      uuidUser           = tranFactory.newArmUUID(uuidUserBytes);

      tranClient     = tranFactory.newArmTransaction(uuidTranClient);
      tranAppServer  = tranFactory.newArmTransaction(uuidTranAppServer);
      tranDataServer = tranFactory.newArmTransaction(uuidTranDataServer);

      random = new Random(System.currentTimeMillis()); // Initialize with a double
   }

   //---------------------------------------------------------------------
   // nextExponential()
   //---------------------------------------------------------------------
   private long nextExponential(long mu)
   {  return (long) (-mu*Math.log(1-random.nextDouble()));
      //use 1-x to get range (0,1] rather than [0,1)
   }
```

```
   //-----------------------------------------------------------------
   // run()
   //-----------------------------------------------------------------
   public void run()
   {
      // Each iteration executes three transactions.
      // - The first requests a correlator.
      // - The second passes the correlator from the first as a parent correlator
and
      //     requests a correlator.
      // - The third passes the correlator from the second as a parent correlator.
      for ( int i = 0; i<numTrans; ++i)
      {
         ArmCorrelator clientCorr, appServerCorr;

         tranClient.start(uuidUser);
         clientCorr = tranClient.getCorr();
         simulateTran();

         tranAppServer.start(uuidUser, clientCorr);
         appServerCorr = tranAppServer.getCorr();
         simulateTran();

         tranDataServer.start(appServerCorr);
         simulateTran();

         tranDataServer.stop(ArmConstants.ARM_GOOD);
         tranAppServer.stop(ArmConstants.ARM_GOOD);
         tranClient.stop(ArmConstants.ARM_GOOD);
      }
   }

   //-----------------------------------------------------------------
   // simulateTran()
   //-----------------------------------------------------------------
   private void simulateTran()
   {
      // Calculate how long the next transaction will be
      sleepTime = nextExponential(msecBtwTrans);
      try
      {  Thread.sleep(sleepTime);
      }
      catch (InterruptedException e)
      {
      }
   }

}//-->ProducerTranCorrUser
```

Appendix B

Information for Implementers

This appendix contains information useful to creators of ARM implementations, and analysis and reporting programs that process ARM data. Applications using ARM to measure transactions do not use any of this information.

B.1 Byte Ordering in Correlators

Correlators are passed from application to application. The transfer may occur within a single system or a single JVM, or it may occur across a network. The recipient and sender of a correlator may run on different machines with different architectures, and the conventions for ordering bytes in data fields, such as integers and arrays, may be different.

If all the programs that touch a correlator were written in Java, the JVM would ensure that the same ordering conventions are followed. Although this is a specification for Java programs, this may not be true. This is the second version of ARM that uses correlators. ARM 2.0 is a C language interface. An ARM 3.0 for C Programs standard will probably be specified soon, as well. Each program will use the version of the standard that makes most sense for it, and for which suitable measurement tools are available. It is unwise to constrain this choice because of the choice made by an application on a different system.

Recognizing this fact, ARM is designed expressly to permit correlators to be exchanged between any application using ARM and any ARM implementation, regardless of how it is written. An application using ARM 3.0 for Java Programs may receive a (parent) correlator from an application using ARM 2.0 (for C programs), and it may send its correlator to an application using ARM 3.0 for C Programs. To permit these types of exchanges, ARM specifies the ordering of bytes within the correlator.

All correlator fields, and the correlator itself, are sent in network byte order. Network byte order is a standard described as follows. The most significant bit is the first bit sent, and the least significant bit is the last bit sent. For example, a 32-bit integer field would be sent with the most significant byte first, and the least significant byte would be the fourth byte sent. Within each byte

Byte 0	Byte 1	Byte 2	Byte 3
0 1 2 3 4 5 6 7	8 9 10 11 12 13 14 15	16 17 18 19 20 21 22 23	24 25 26 27 28 29 30 31
Bit 0 is the most significant bit			Bit 31 is the least significant bit

B.2 Correlator Formats

ARM specifies formatting constraints that all correlators must adhere to. In addition, ARM defines three specific correlator formats. Other formats may be added in the future.

The three defined correlator formats are listed here, and each is described in detail in the following sections.

- In the ARM 2.0 correlator format, the transaction ID and handle are 32-bit integers. The correlator format byte has a value of '1'.
- ARM 3.0 adds a new correlator format that is the same as the ARM 2.0 correlator format except that the transaction ID is a 16-byte UUID and the handle is a 64-bit integer. The correlator format byte has a value of '2'.
- ARM 3.0 adds a new correlator format that is used by an ARM implementation when it is asked to create a correlator but it cannot recognize the data. One likely scenario of this type is when the `ArmTransactionFactory` method `newArmCorrelator()` is executed and the byte array parameter is null or the offset value is incorrect, such as extending past the end of the array. Another likely scenario is when the byte array passed to `newArmCorrelator()` contains an invalid length in the first two bytes, such as zero. The correlator format byte has a value of '127'.

ARM also specifies a range of correlator format IDs that are available for implementers of ARM to use for proprietary formats. The range is (in unsigned integer notation) 128:255 or (in signed integer notation) –128:–1 or (in hexadecimal notation) x'80'-x'FF'. There is a potential for collisions if two implementers use the same format value. To lessen the probability that this will occur, this specification lists known values, though the specification does not limit how vendors may use the values in this range. One known value is used at this time:

- 128 (unsigned), –128 (signed), or x'80' (hexadecimal) is used by Hewlett-Packard.

B.3 ARM Correlator Format Constraints

These constraints apply to all formats.

Table 1. ARM Correlator Format Constraints

Position	Length	Contents
Bytes 0:1	2 bytes	Length of the correlator, including these two bytes. Valid lengths are 4 <= length <= 168. • Lengths shorter than 4 bytes are not permitted because all correlators must have the four bytes defined in this table. • Lengths longer than 168 bytes are not permitted because they would overflow a buffer in the ARM 2.0 C API call `arm_start()`.
Byte 2	1 byte	Correlator format The range 0:127 (unsigned) is reserved by the standard. The range 128:255 (unsigned) is available for use by ARM implementers.
Byte 3	1 byte	Flags All eight bit flags are reserved by the standard. Two flags are defined in positions 0:1 (the highest order bits), as in ab000000, where a and b are bit flags. a = 1 if a trace of this transaction is requested by the agent that generated the correlator. This is transparent to the applications. b = 1 if the application indicates that this transaction is of particular importance, such as a test transaction, and therefore worthy of being traced. This was a function in ARM 2.0 that was removed in ARM 3.0. There are no requirements for how these flags are handled, if at all. The usage scenario that led to their creation was to enable a trace of selected transactions throughout an enterprise. A selective trace would yield much useful information without being a significant burden on the systems processing the transaction. For example, a client could be experiencing response time problems. The agent on the client could turn on the trace flag (bit 0) in the correlators that it generates. When this correlator is passed, as the parent correlator, to the ARM implementation on the server, the ARM implementation could turn on the trace flag in the correlators that it generates. The process could continue recursively. What has

Position	Length	Contents
		resulted is a trace of all the transactions associated with the client experiencing the response time problem, but only those transactions. If there are 1,000 clients in the enterprise running this application, 0.1% of all transactions are traced, which is a minimal load on the systems. The value of a surgical trace like this was considered great enough to justify including it in the standard.

B.4 ARM Correlator Format 1 (defined in ARM 2.0)

Table 2. ARM Correlator Format 1

Position	Length	Contents
Bytes 0:1	2 bytes	Length of the correlator, including these two bytes.
Byte 2	1 byte	Correlator format = 1
Byte 3	1 byte	Flags See the description in Table 1. ARM Correlator Format Constraints on page 90.
Bytes 4:5	2 bytes	Format of the address field (signed 16-bit). The following formats are defined: 0 = reserved 1 = IP version 4 (4 bytes) 2 = IP version 4 plus a port number (6 bytes) 3 = IP version 6 (16 bytes) 4 = IP version 6 plus a port number (18 bytes) 5 = SNA (IBM's System Network Architecture) (16 bytes) 6 = X.25 (16 bytes) 7 = hostname (variable length) 8:32767 = reserved -32768:-1 = undefined and available for agent implementers to use. There are no semantics associated with the address format. It will be an unusual situation where a new format is needed, but this provides a solution if this is needed. The preferred approach is to get a new format defined that is in the 0-32767 range. There is a risk that two different agent developers will choose the same id, but this risk is deemed small. Note that this field was originally designated as an unsigned field. Because Java does not support an unsigned 16-bit integer, but only a signed 16-bit integer ("short"), the definition has been changed to a signed 16-bit integer. This does not restrict the range in any way. It changes the notation. The original specification assigned the range of available and unreserved formats to be 32768-65535. The equivalent signed range is –32768 to –1 (all the negative values).

Position	Length	Contents
Bytes 6:7	2	Vendor ID (signed 16-bit) The vendor ID is a way to identify who built the agent. Combining this information with the Agent Version field will provide a way for a management application to know what kind of agent generated a correlator. A management application may contain specialized functions or logic that only works with the agents from a particular vendor and/or supporting particular functions or interfaces. By putting these two fields in the correlator, a management application has a way to know whether the agent that generated the correlator has some of these specialized capabilities. In order to minimize the possibility of two vendors using the same vendor ID, the value should be taken from the list of enterprise identifiers from the Internet Assigned Numbers Authority (IANA). This list was created for vendors who have SNMP agents. Although the ARM API specification does not require or endorse SNMP, it's likely that most or all the organizations that will create an ARM agent will have at least one enterprise ID assigned. The list of enterprise IDs can be found at: ftp://ftp.isi.edu/in-notes/iana/assignments/enterprise-numbers For organizations that don't have an enterprise identifier assigned by the IANA, the negative values are free for agent developers to use. There are no semantics associated with these ids. It is expected that most or all agent developers will have a formally assigned vendor id, and it will be an unusual situation where another id is needed, but this provides a solution if this is needed. There is a risk that two different agent developers will choose the same id, but this risk is deemed small. Note that this field was originally designated as an unsigned field. Because Java does not support an unsigned 16-bit integer, but only a signed 16-bit integer (“short”), the definition has been changed to a signed 16-bit integer. This does not restrict the range in any way. It changes the notation. The original specification assigned the range of available and unreserved formats to be 32768-65535. The equivalent signed range is –32768 to –1 (all the negative values).
Bytes 8:9	2	Agent version (signed 16-bit) The Agent Version is used to distinguish between different versions of an agent, and will be most useful when the capabilities and/or interfaces of an agent change from one release to another. It will also be useful to distinguish between different agents from the same vendor. Each vendor is responsible for avoiding having multiple

Position	Length	Contents
		agents with different capabilities using the same Agent Version value. Note that this field was originally designated as an unsigned field. Because Java does not support an unsigned 16-bit integer, but only a signed 16-bit integer ("short"), the definition has been changed to a signed 16-bit integer. This does not restrict the range in any way.
Bytes 10:11	2	Agent instance (signed 16-bit) There are several scenarios in which it is possible for multiple agents (ARM implementations) to be running simultaneously on the same system, each generating handles independently of all the others. To avoid duplicate correlators, each agent should be assigned a unique value, a value that does not conflict with any other agent running on the system. Note that this field was originally designated as an unsigned field. Because Java does not support an unsigned 16-bit integer, but only a signed 16-bit integer ("short"), the definition has been changed to a signed 16-bit integer. This does not restrict the range in any way.
Bytes 12:15	4	Transaction handle (32-bit integer) In ARM 2.0, this is the `start_handle` returned from an `arm_start()` call.
Bytes 16:19	4	Transaction ID (32-bit integer) In ARM 2.0, this is the `tran_id` returned from an `arm_getid()` call.
Bytes 20:21	2	Length of the system ID field (signed 16-bit) Note that this field was originally designated as an unsigned field. Because Java does not support an unsigned 16-bit integer, but only a signed 16-bit integer ("short"), the definition has been changed to a signed 16-bit integer. This does not restrict the range in a meaningful range because the maximum size of the field is 146 bytes.
Bytes 22:167	varies from 1-146 bytes	System ID (byte array) This is the network address or hostname of the system on which the transaction executed, or a proxy that contains the actual information. The format used here must match the format defined in Bytes 4:5. When the length varies, the length of the system ID field determines how long the field is. The following formats are defined: 0 = reserved

Position	Length	Contents
		1 = IP version 4 (4 bytes) Bytes 0:3 = 4-byte IP address 2 = IP version 4 plus a port number (6 bytes) Bytes 0:3 = 4-byte IP address Bytes 4:5 = 2-byte IP port number 3 = IP version 6 (16 bytes) Bytes 0:15 = 16-byte IPv6 address 4 = IP version 6 plus a port number (18 bytes) Bytes 0:15 = 16-byte IPv6 address Bytes 16:17 = 2-byte IP port number 5 = SNA (IBM's System Network Architecture) (16 bytes) Bytes 0:7 = 8-byte EBCDIC-encoded network ID Bytes 8:15 = 8-byte EBCDIC-encoded network accessible unit (control point or LU) 6 = X.25 (16 bytes) Bytes 0:15 = The X.25 network address, also referred to as an X.121 address. This is up to sixteen ASCII character digits ranging from 0:9. The length is known from the "Length of the System ID field" in Bytes 20:21. 7 = hostname (variable length) Bytes 0:?? The length is known from the "Length of the System ID field" in Bytes 20:21.

B.5 ARM Correlator Format 2 (defined in ARM 3.0)

Table 3. ARM Correlator Format 2

Position	Length	Contents
Bytes 0:1	2 bytes	Length of the correlator, including these two bytes.
Byte 2	1 byte	Correlator format = 2
Byte 3	1 byte	Flags See the description in Table 1. ARM Correlator Format Constraints on page 90.
Bytes 4:5	2 bytes	Format of the address field (signed 16-bit). The following formats are defined: 0 = reserved 1 = IP version 4 (4 bytes) 2 = IP version 4 plus a port number (6 bytes) 3 = IP version 6 (16 bytes) 4 = IP version 6 plus a port number (18 bytes) 5 = SNA (IBM's System Network Architecture) (16 bytes) 6 = X.25 (16 bytes) 7 = hostname (variable length) 8:32767 = reserved -32768:-1 = undefined and available for agent implementers to use. There are no semantics associated with the address format. It will be an unusual situation where a new format is needed, but this provides a solution if this is needed. The preferred approach is to get a new format defined that is in the 0-32767 range. There is a risk that two different agent developers will choose the same id, but this risk is deemed small. Note that this field was originally designated as an unsigned field. Because Java does not support an unsigned 16-bit integer, but only a signed 16-bit integer ("short"), the definition has been changed to a signed 16-bit integer. This does not restrict the range in any way. It changes the notation. The original specification assigned the range of available and unreserved formats to be 32768-65535. The equivalent signed range is –32768 to –1 (all the negative values).

Position	Length	Contents
Bytes 6:7	2	Vendor ID (signed 16-bit) The vendor ID is a way to identify who built the agent. Combining this information with the Agent Version field will provide a way for a management application to know what kind of agent generated a correlator. A management application may contain specialized functions or logic that only works with the agents from a particular vendor and/or supporting particular functions or interfaces. By putting these two fields in the correlator, a management application has a way to know whether the agent that generated the correlator has some of these specialized capabilities. In order to minimize the possibility of two vendors using the same vendor ID, the value should be taken from the list of enterprise identifiers from the Internet Assigned Numbers Authority (IANA). This list was created for vendors who have SNMP agents. Although the ARM API specification does not require or endorse SNMP, it's likely that most or all the organizations that will create an ARM agent will have at least one enterprise ID assigned. The list of enterprise IDs can be found at: ftp://ftp.isi.edu/in-notes/iana/assignments/enterprise-numbers For organizations that don't have an enterprise identifier assigned by the IANA, the negative values are free for agent developers to use. There are no semantics associated with these ids. It is expected that most or all agent developers will have a formally assigned vendor id, and it will be an unusual situation where another id is needed, but this provides a solution if this is needed. There is a risk that two different agent developers will choose the same id, but this risk is deemed small. Note that this field was originally designated as an unsigned field. Because Java does not support an unsigned 16-bit integer, but only a signed 16-bit integer (“short”), the definition has been changed to a signed 16-bit integer. This does not restrict the range in any way. It changes the notation. The original specification assigned the range of available and unreserved formats to be 32768-65535. The equivalent signed range is –32768 to –1 (all the negative values).
Bytes 8:9	2	Agent version (signed 16-bit) The Agent Version is used to distinguish between different versions of an agent, and will be most useful when the capabilities and/or interfaces of an agent change from one release to another. It will also be useful to distinguish between different agents from the same vendor. Each vendor is responsible for avoiding having multiple

Position	Length	Contents
		agents with different capabilities using the same Agent Version value. Note that this field was originally designated as an unsigned field. Because Java does not support an unsigned 16-bit integer, but only a signed 16-bit integer ("short"), the definition has been changed to a signed 16-bit integer. This does not restrict the range in any way.
Bytes 10:11	2	Agent instance (signed 16-bit) There are several scenarios in which it is possible for multiple agents (ARM implementations) to be running simultaneously on the same system, each generating handles independently of all the others. To avoid duplicate correlators, each agent should be assigned a unique value, a value that does not conflict with any other agent running on the system. Note that this field was originally designated as an unsigned field. Because Java does not support an unsigned 16-bit integer, but only a signed 16-bit integer ("short"), the definition has been changed to a signed 16-bit integer. This does not restrict the range in any way.
Bytes 12:19	8	Transaction handle (64-bit integer) The format of the handle is determined by the ARM implementation. There are no assumed semantics because it is used as an identifier. The requirement is that it be unique within the system ID plus agent instance, for the longest possible expected lifetime of a transaction record. ARM implementations may find it advantageous to use a format that is not simply a rolling counter, but is based on some other structure. For example, an SDK (software developers kit) used to test the ARM 3.0 for Java Programs specification used a format that combined 44 bits from the Java system clock with a 20-bit rolling counter. This provided the time of day when a transaction executed, as well as the uniqueness required by the standard.
Bytes 20:35	16	Transaction UUID (16-byte UUID)
Bytes 36:37	2	Length of the system ID field (signed 16-bit) Note that this field was originally designated as an unsigned field. Because Java does not support an unsigned 16-bit integer, but only a signed 16-bit integer ("short"), the definition has been changed to a signed 16-bit integer. This does not restrict the range in a meaningful range because the maximum size of the field is 146 bytes.
Bytes 38:167	varies from	System ID (byte array)

Position	Length	Contents
	1-130 bytes	This is the network address or hostname of the system on which the transaction executed, or a proxy that contains the actual information. The format used here must match the format defined in Bytes 4:5. When the length varies, the length of the system ID field determines how long the field is. The following formats are defined: 0 = reserved 1 = IP version 4 (4 bytes) Bytes 0:3 = 4-byte IP address 2 = IP version 4 plus a port number (6 bytes) Bytes 0:3 = 4-byte IP address Bytes 4:5 = 2-byte IP port number 3 = IP version 6 (16 bytes) Bytes 0:15 = 16-byte IPv6 address 4 = IP version 6 plus a port number (18 bytes) Bytes 0:15 = 16-byte IPv6 address Bytes 16:17 = 2-byte IP port number 5 = SNA (IBM's System Network Architecture) (16 bytes) Bytes 0:7 = 8-byte EBCDIC-encoded network ID Bytes 8:15 = 8-byte EBCDIC-encoded network accessible unit (control point or LU) 6 = X.25 (16 bytes) Bytes 0:15 = The X.25 network address, also referred to as an X.121 address. This is up to sixteen ASCII character digits ranging from 0:9. The length is known from the "Length of the System ID field" in Bytes 20:21. 7 = hostname (variable length) Bytes 0:?? The length is known from the "Length of the System ID field" in Bytes 20:21.

B.6 ARM Correlator Format 127 (defined in ARM 3.0)

Table 4. ARM Correlator Format 127

Position	Length	Contents
Bytes 0:1	2 bytes	Length of the correlator, including these two bytes.
Byte 2	1 byte	Correlator format = 127
Byte 3	1 byte	Flags See the description in Table 1. ARM Correlator Format Constraints on page 90.
Bytes 4:167	varies from 0-164 bytes	Data There are no constraints on what may be in this field.

Index

A full index will be added in the published version of the Standard.