Technical Standard

Application Response Measurement (ARM)

Issue 4.1 Version 1 – Java Binding

Technical Standard

Application Response Measurement (ARM) Issue 4.0 Version 1 – Java Binding

ISBN: 1-931624-75-5

Document Number: C072

Published by The Open Group, April 2007.

Comments relating to the material contained in this document may be submitted to:

The Open Group, Thames Tower, 37-45 Station Road, Reading, Berkshire, RG1 1LX, United Kingdom

or by electronic mail to: ospecs@opengroup.org

Contents

Preface

The Open Group

The Open Group is a vendor-neutral and technology-neutral consortium, whose vision of Boundaryless Information Flow™ will enable access to integrated information within and between enterprises based on open standards and global interoperability. The Open Group works with customers, suppliers, consortia, and other standards bodies. Its role is to capture, understand, and address current and emerging requirements, establish policies, and share best practices; to facilitate interoperability, develop consensus, and evolve and integrate specifications and Open Source technologies; to offer a comprehensive set of services to enhance the operational efficiency of consortia; and to operate the industry's premier certification service, including UNIX® certification.

Further information on The Open Group is available at www.opengroup.org.

The Open Group has over 15 years' experience in developing and operating certification programs and has extensive experience developing and facilitating industry adoption of test suites used to validate conformance to an open standard or specification.

More information is available at www.opengroup.org/certification.

The Open Group publishes a wide range of technical documentation, the main part of which is focused on development of Technical and Product Standards and Guides, but which also includes white papers, technical studies, branding and testing documentation, and business titles. Full details and a catalog are available at www.opengroup.org/bookstore.

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.

Readers should note that updates – in the form of Corrigenda – may apply to any publication. This information is published at www.opengroup.org/corrigenda.

This Document

This document is the Technical Standard for the Java Binding for Application Response Measurement (ARM) Issue 4.1. It has been developed and approved by The Open Group.

ARM is a standard for measuring service levels of single-system and distributed applications. ARM measures the availability and performance of transactions, both visible to the users of the business application and those visible only within the IT infrastructure.

Typographical Conventions

The following typographical conventions are used throughout this document:

- Arial font is used in text for Java elements.
- Arial Bold font is used in text for Java methods.
- Syntax and code examples are shown in fixed width font.

Trademarks

Boundaryless Information Flow™ and TOGAF™ are trademarks and Making Standards Work®, The Open Group®, and UNIX® are registered trademarks of The Open Group in the United States and other countries.

Hewlett-Packard® is a registered trademark of Hewlett-Packard Company.

Java® is a registered trademark of Sun Microsystems, Inc.

Tivoli™ is a trademark of Tivoli Systems, Inc.

WebSphere® is a registered trademark of IBM Corporation.

The Open Group acknowledges that there may be other brand, company, and product names used in this document that may be covered by trademark protection and advises the reader to verify them independently.

Acknowledgements

The Open Group gratefully acknowledges the contribution of the following people in the development of this document:

- Bill Furnas, Hewlett-Packard
- Mark Johnson, IBM
- Marcus Thoss, tang-IT Consulting GmbH
- Van Wiles, BMC

Referenced Documents

The following documents are referenced in this Technical Standard:

ARM 2.0 Technical Standard, July 1998, Systems Management: Application Response Measurement (ARM) (C807), published by The Open Group.

ARM 3.0 Technical Standard, October 2001, Application Response Measurement (ARM) Issue 3.0 Java Binding (C014), published by The Open Group.

ARM 4.0 (C Binding)
Technical Standard, August 2004, Application Response Measurement (ARM) Issue 4.0 Version 2 – C Binding (C041), published by The Open Group.

ARM 4.1 (C Binding)
Technical Standard, April 2007, Application Response Measurement (ARM) Issue 4.1 Version 1 – C Binding (C071), published by The Open Group.

IEEE Std 754-1985
IEEE Standard for Binary Floating-Point Arithmetic.

Unicode (UCS)
Refer to www.unicode.org.

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 are 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.

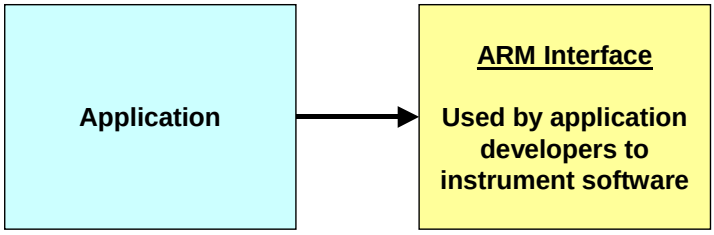

Figure 1: Application – ARM Interface

From the system administrator's perspective, ARM consists of the interfaces that applications load and call and 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 specification, but it is, of course, important to the system administrator.

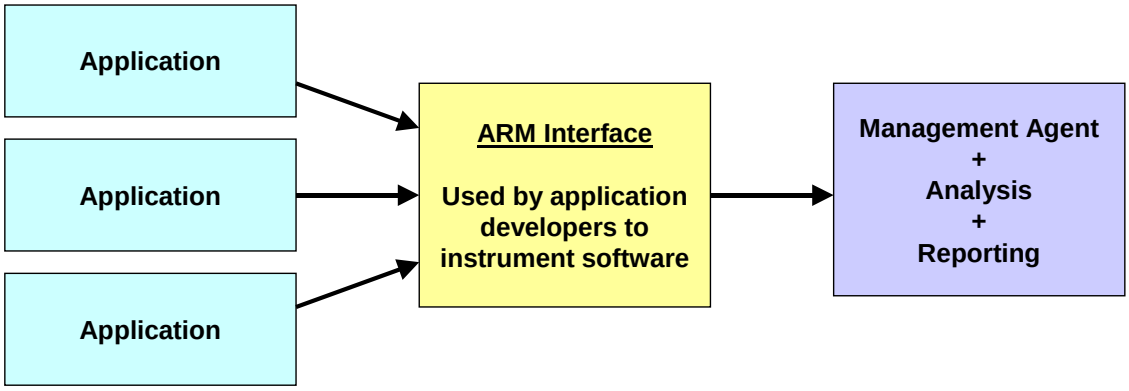

Figure 2: Application – ARM Management System Interaction

1.3 Selecting Transactions to Measure

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 Technical Standard of The Open Group 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 was developed by The Open Group in 2001. It added new capabilities and specified interfaces in the Java programming language. ARM 3.0 added the following capabilities:

- Java bindings
- Changes to the length of application and transaction identifiers and handles
- The ability to report the status and response time of a transaction with a single call, executed after the transaction completes; the transaction could have executed on a different system
- The ability to identify a user on a per-transaction basis

ARM 4.0 was developed to implement new capabilities, and to provide equivalent functions for both C and Java programs. ARM 4.0 added the following capabilities:

- A richer and more flexible model for specifying application and transaction identity
- Report attributes of a transaction that change on a per-instance basis
- Bind a transaction to a thread
- Indicate the amount of time a transaction is blocked waiting for an external event
- Indicate the true time when a transaction started executing for a specialized situation in which the standard indication of a start [*arm_start_transaction*() in ARM 4.0 C bindings] will not yield an accurate time

ARM 4.1 has been developed to implement new capabilities while maintaining equivalent functions for both C and Java programs. This document describes the Java program bindings. A

companion document describes the C program bindings. ARM 4.1 adds the following capabilities:

- An instrumentation control interface that an application may use to query an implementation to determine how much information about a transaction would be useful (from none to very detailed)
- Interfaces to improve ARM's usefulness in messaging and workflow environments
- More flexible mechanisms for reporting properties about a transaction; this is especially useful in instrumented middleware that may not know the properties about a transaction of an application that runs on the middleware until the transaction executes
- Additional flexibility to report when transactions begin executing

1.5 Compatibility between ARM Java Binding Versions

ARM defines low-level programming interfaces to be used between programs that are dynamically linked together. This limits how much an interface can change from version to version and still link with programs using a previous version. In particular, changing function entry points, or the call signatures of an entry point, prevents working with a different version.

There are three versions of ARM Java bindings.

- ARM 3.0 is incompatible with ARM 4.0 and ARM 4.1. It is expected that management agents may simultaneously support multiple versions of ARM. For example, a product may support both the ARM 3.0 and ARM 4.0 Java interfaces. However, a business application using the ARM 3.0 interfaces cannot instantiate the ARM 4.0 interfaces, and *vice versa*.
- Applications using ARM 4.0 can interoperate with implementations of either ARM 4.0 or ARM 4.1.
- Applications using ARM 4.1 can interoperate with implementations of ARM 4.1. If the application restricts the calls it makes to those supported in ARM 4.0, even though it has compiled using the ARM 4.1 interface files, then the application can interoperate with an ARM 4.0 implementation.

1.6 ARM 4.0 Java Bindings Overview

This specification describes three Java packages. Each package is equivalent to the block titled "ARM Interface" in Figure 1 and Figure 2.

- org.opengroup.arm40.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.arm40.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.arm40.metric can be used in addition to the org.opengroup.arm40.transaction package to report additional measurements about each transaction, such as a count of the amount of work accomplished.

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 (Java Virtual Machine) process to be analyzed and reported.

1.7 Terminology

The following terminology is used throughout this document:

Can — Describes a permissible optional feature or behavior available to the user or application. The feature or behavior is mandatory for an implementation that conforms to this document. An application can rely on the existence of the feature or behavior.

Implementation-defined — (Same meaning as "implementation-dependent".) Describes a value or behavior that is not defined by this document but is selected by an implementer. The value or behavior may vary among implementations that conform to this document. An application should not rely on the existence of the value or behavior. An application that relies on such a value or behavior cannot be assured to be portable across conforming implementations. The implementer shall document such a value or behavior so that it can be used correctly by an application.

May — Describes a feature or behavior that is optional for an implementation that conforms to this document. An application should not rely on the existence of the feature or behavior. An application that relies on such a feature or behavior cannot be assured to be portable across conforming implementations. To avoid ambiguity, the opposite of "may" is expressed as "need not", instead of "may not".

Must — Describes a feature or behavior that is mandatory for an application or user. An implementation that conforms to this document shall support this feature or behavior.

Shall — Describes a feature or behavior that is mandatory for an implementation that conforms to this document. An application can rely on the existence of the feature or behavior.

Should — For an implementation that conforms to this document, describes a feature or behavior that is recommended but not mandatory. An application should not rely on

the existence of the feature or behavior. An application that relies on such a feature or behavior cannot be assured to be portable across conforming implementations. For an application, describes a feature or behavior that is recommended programming practice for optimum portability.

Undefined Describes the nature of a value or behavior not defined by this document that results from use of an invalid program construct or invalid data input. The value or behavior may vary among implementations that conform to this document. An application should not rely on the existence or validity of the value or behavior. An application that relies on any particular value or behavior cannot be assured to be portable across conforming implementations.

Unspecified Describes the nature of a value or behavior not specified by this document that results from use of a valid program construct or valid data input. The value or behavior may vary among implementations that conform to this document. An application should not rely on the existence or validity of the value or behavior. An application that relies on any particular value or behavior cannot be assured to be portable across conforming implementations.

Will Same meaning as “shall”; “shall” is the preferred term.

2 Getting Started Using ARM 4.0/4.1 Java Bindings

ARM is designed to be a relatively simple interface with several optional capabilities. Many applications need only the essential elements – perhaps augmented by a small subset of the optional capabilities – to satisfy their instrumentation requirements.

In order to describe all of ARM's optional capabilities, this specification is by necessity large. Most users need to understand only a subset of the specification. This chapter describes three common sets of capabilities and directs the reader to the relevant sections in the specification. It is hoped this will assist users to quickly start using ARM.

2.1 Basic End-to-End Measurements

All applications must use or are recommended to use the following capabilities:

Capability	Suggested Reading	Mandatory Interfaces (Alphabetically)	Optional Interfaces (Alphabetically)
Registration and initialization	Chapter 5 Chapter 11	ArmApplication ArmApplicationDefinition ArmID ArmTransactionDefinition ArmTransactionFactory	ArmIdentityProperties ArmIdentityPropertiesTransaction
Measure response time and status	Chapter 3	ArmTransaction: • **start()** • **stop()**	
[optional] Correlate transactions end-to-end	Section 4.1	ArmCorrelator ArmTransaction: • **getCorrelator()** • **getParentCorrelator()**	
There is a code sample in Section A.1.			

Table 1 ARM Capabilities for Basic End-to-End Measurements

2.2 Detailed Timing and Threading Measurements

The blocking and threading capabilities described below are useful with management software that monitors and/or manages how transactions interact with the operating system, such as workload management software.

The response time detail measurements are useful when there are transactions that begin executing prior to the time when all the identity context data for the transaction is known.

The best practice instrumentation uses the capabilities described in Table 2 in addition to the basic capabilities described in Table 1.

Capability	Suggested Reading	Mandatory Interfaces	Optional Interfaces
[optional] Measure time that transactions are blocked	Section 6.2	ArmTransaction: • **blocked()** • **unblocked()**	ArmBlockCause
[optional] Associate transactions to threads	Section 6.3	ArmTransaction: • **bindThread()** • **setAutomaticBindThread()** • **unbindThread()**	
[optional] Measure response times more accurately in certain situations	Section 6.1	ArmTransaction: • **setArrivalTime()** • **setPrestartTimeValue()**	ArmPrestartTimeStats ArmTimestamp
There is a code sample in Section A.2.			

Table 2 ARM Capabilities for Detailed Timing and Threading Measurements

2.3 Messaging

The message instrumentation capabilities described below are useful with transactions that use messages to be invoked and/or to invoke other transactions. The best practice instrumentation uses the capabilities described in Table 3 in addition to the basic capabilities described in Table 1.

Capability	Suggested Reading	Mandatory Interfaces	Optional Interfaces
[optional] Indicate message sent and received events	Section 4.2	ArmMessageEventGroup ArmMessageReceivedEvent ArmMessageSentEvent ArmTransaction: • **setMessageEventGroup()**	
There is a code sample in Section A.3.			

Table 3 ARM Capabilities for Messaging Instrumentation

3 Programming Options

The application has two options for providing measurement data:

- In Option 1, the preferred and more widely used option, the application calls an ArmTransaction interface just before and after a transaction executes, and the ArmTransaction object makes the measurements.
- In Option 2, the application makes all the measurements itself and reports the data some time later. Option 2 should only be used in situations that preclude using Option 1.

3.1 ARM Measures Transactions

Figure 3 shows Option 1, the preferred and most widely used option. 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 re-use the ArmTransaction instance. The data will have already been copied from it to be processed.

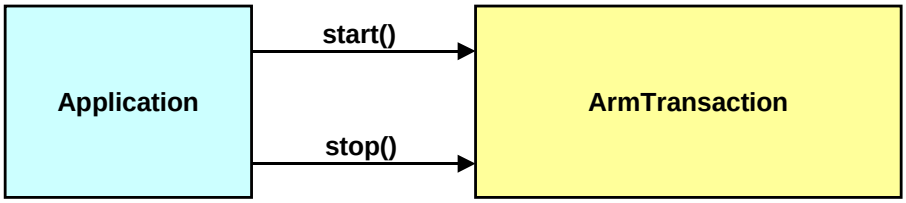

Figure 3: Measurement using Start/Stop

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.

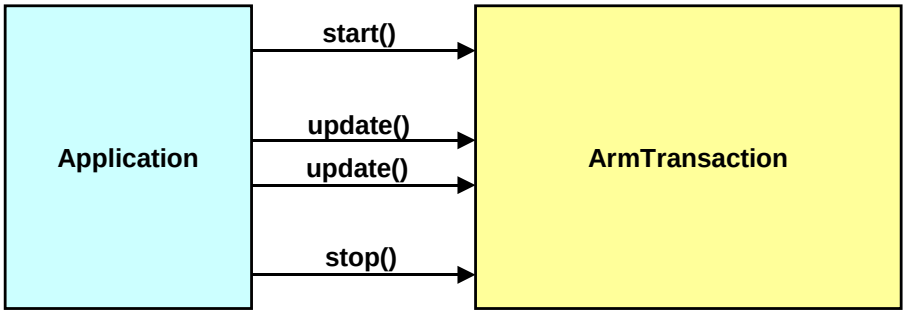

Figure 4: Application using Heartbeats

3.2 Application Measures Transactions

Figure 5 shows Option 2. The application itself measures the response time of the transaction. After the transaction completes (the delay could be short or long), it populates an ArmTranReport object with data, and calls **report()** to initiate processing of the data. As soon as the **report()** method returns, the application is free to reuse the ArmTranReport instance. The data will have already been copied from it to be processed.

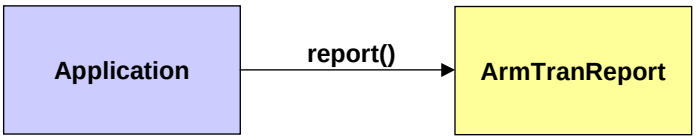

Figure 5: Measurement by the Application

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 Option 1 (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 are used. The calls must be made at the moment the real transaction starts and stops. If they are not, 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 (Java Virtual Machine) instance of the remote system where the transaction ran.

4 Understanding the Relationships between Transactions

There are several solutions available that measure transaction response times, 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.

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

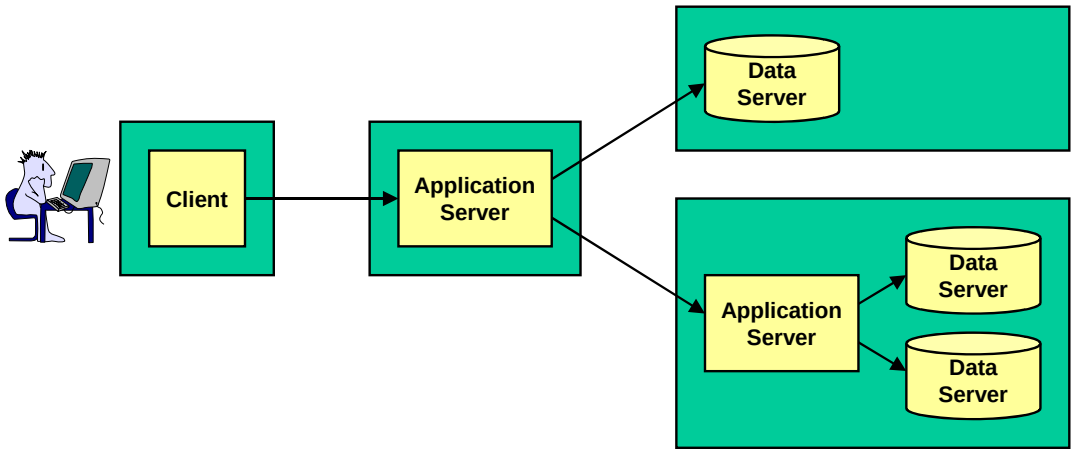

Figure 6: A Common Distributed Application Architecture

4.1 Distributed Transactions with Synchronous Flows

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.

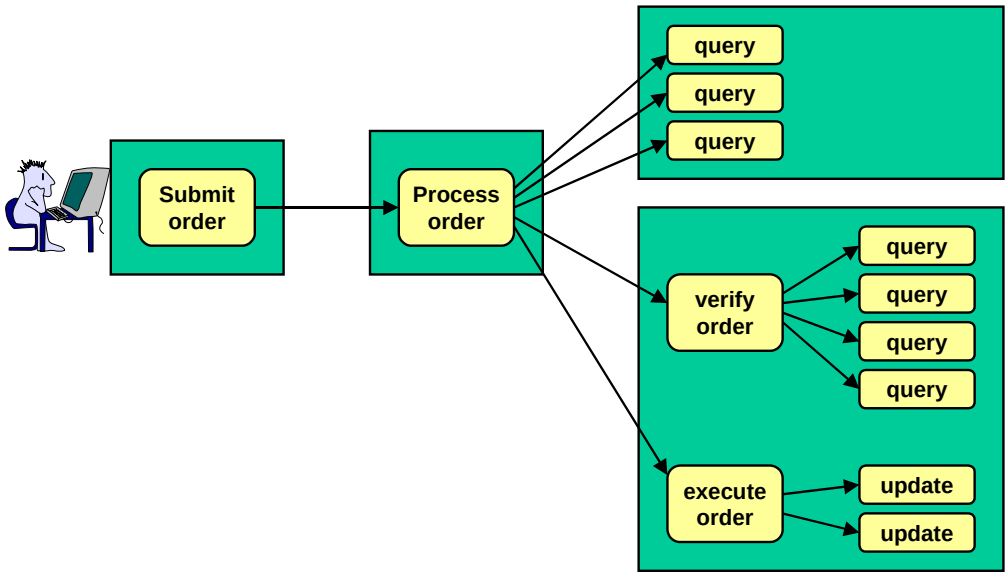

Figure 7: An Example of a Distributed Transaction

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. The correlator format is known to the implementation that creates it, 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.

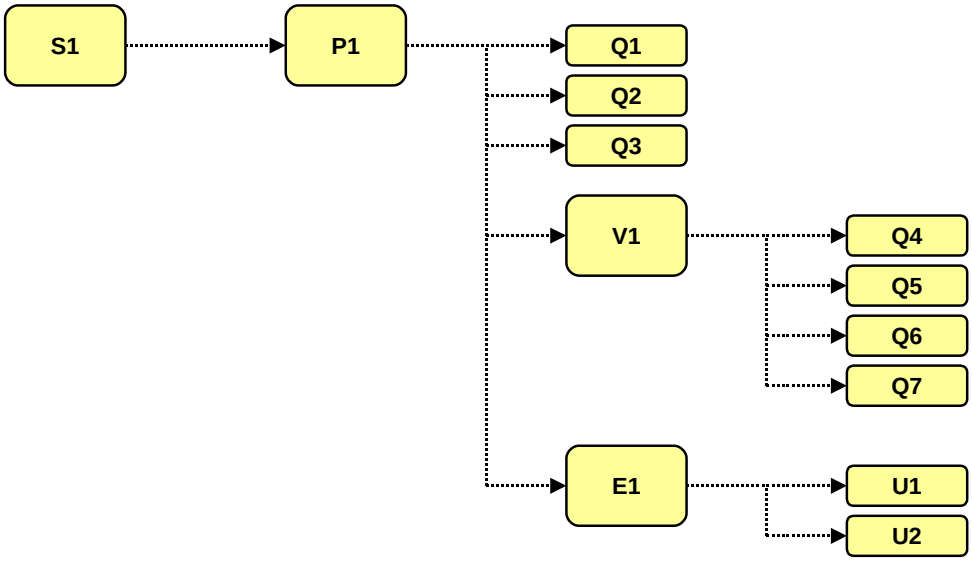

Figure 8: Distributed Transactions that Appear Unrelated

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 transactions, 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 is 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.

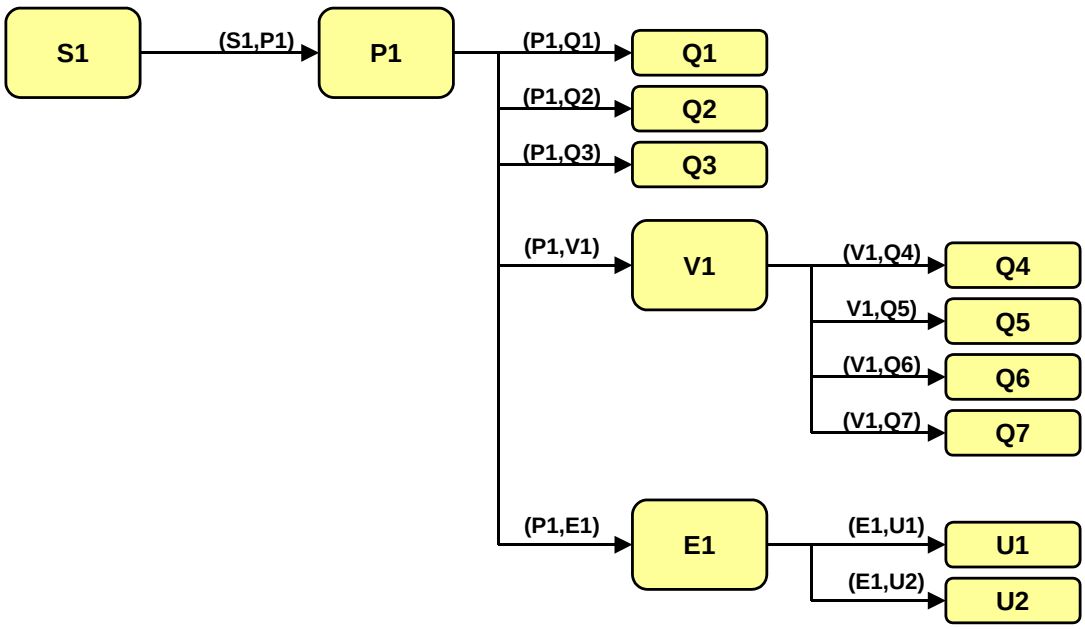

Figure 9: A Distributed Transaction Calling Hierarchy

4.2 Distributed Transactions with Asynchronous Flows (ARM 4.1)

The discussion in Section 4.1 implies synchronous flows between all the component transactions of the distributed transactions. Asynchronous flows using a queuing system are another widely used model for distributed transactions.

An asynchronous flow is one that is accomplished using an asynchronous mechanism, such as a messaging protocol OR if the relationship between the transactions is asynchronous in nature. In the latter case, a parent transaction might initiate a service using a synchronous protocol, but if the parent continues processing other logic in parallel with the invoked service then the overall relationship between the transactions is asynchronous in nature.

In the examples from Figure 10 through Figure 14 the yellow boxes with a single letter in them represent executing transactions. The transactions communicate with each other using queues.

- Figure 10 represents a simple request/response flow. The business logic of A and B are the same as they would be with synchronous flows, but the communication mechanisms are message queues.
- Figure 11 represents a simple request/response flow from the perspective of B or C, each using message queues for communication. From the perspective of A there are two parallel flows that are in a race condition – it is not possible for A to predict in advance whether B or C will return a response first.
- Figure 12 represents a daisy chain flow in which each component passes the results of the transaction on to another component. The last component returns a message to A, which initiated the first flow. Also shown is an optional independent flow (to D) that is spawned by the main flow but which is independent after it is spawned.
- Figure 13 represents a sequential workflow in which no results are returned to the originating transaction (A). This is a common semantic with batch jobs.
- Figure 14 represents a workflow similar to the one in Figure 13, but with parallel paths that converge. C represents a synchronization point, waiting for input from both B and D before proceeding to execute.

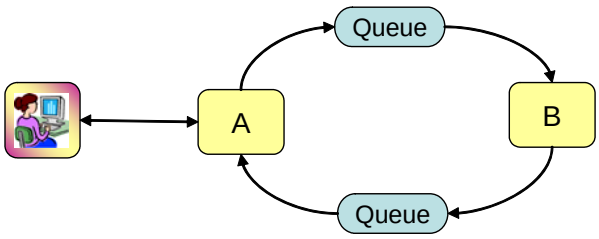

Figure 10: Request/Response with Asynchronous Flows

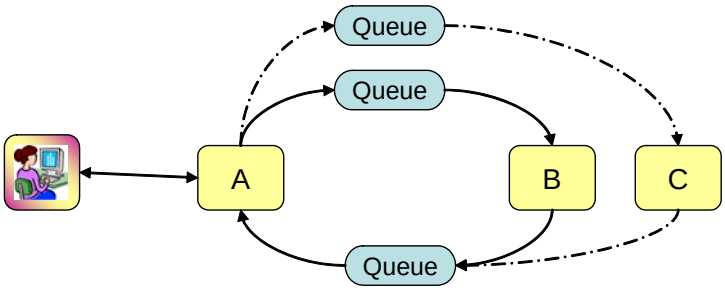

Figure 11: Request/Response with Parallel Asynchronous Flows

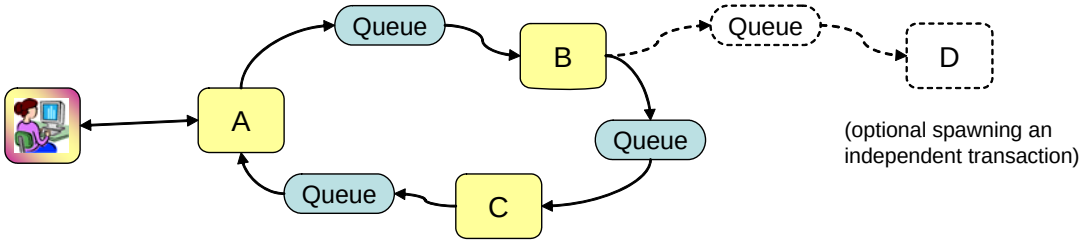

Figure 12: Asynchronous "Daisy-Chain" Asynchronous Flows

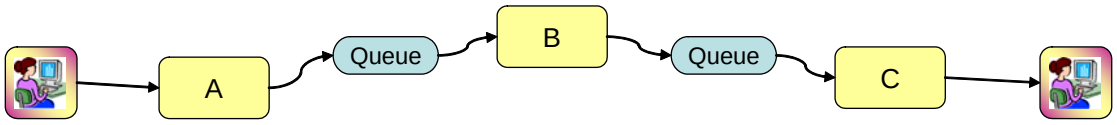

Figure 13: Workflow with Sequential Asynchronous Flows

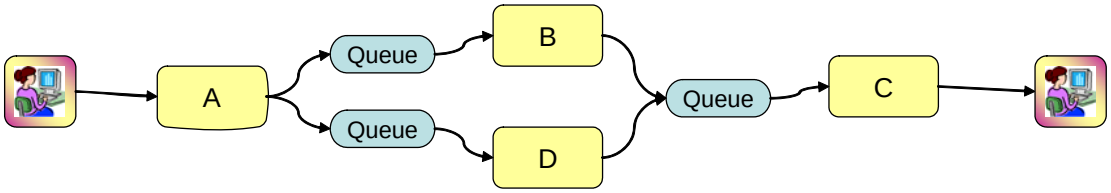

Figure 14: Workflow with Parallel Asynchronous Flows

In each of these five cases it is reasonable for a management application to wish to construct a call graph along the lines of the one depicted in Figure 9. The basic mechanism that is used is the same – creating (parent correlator, current correlator) pairs and then analyzing them to

construct the call graph. ARM 4.1 adds extensions that applications can use to make it clearer that the flows are asynchronous and to understand when a transaction is represented by a circular flow, such as is shown in Figure 12, or a one-way flow, such as is shown in Figure 13.

ARM 4.1 adds three features that enable applications to indicate conditions that describe the relationship between a parent transaction and transactions that it invokes:

- Section 4.2.1 describes the Asynchronous Flow flag in the correlator that indicates that there is an asynchronous relationship between a parent transaction and transactions that it invokes. This flag can be set with the ArmCorrelator **setAsynchronous()** method.
- Section 4.2.2 describes the Independent Transaction flag in the correlator that indicates that an invoked transaction does not influence its parent transaction. This flag can be set with the ArmCorrelator **setIndependentTran()** method.
- Section 4.2.3 describes ArmMessageEvent types and ArmMessageEventGroup that can be used to provide additional details about messages that have been exchanged.

4.2.1 Indicating Asynchronous Flows

Setting the Asynchronous Flow flag to TRUE indicates that the control flow from the parent was accomplished via an asynchronous mechanism, such as a messaging protocol OR that the transaction flow is asynchronous in nature. For example, it would be appropriate to set the Asynchronous Flow flag in the following situation even though synchronous protocols are being used:

- A parent uses a synchronous web service call to send a message to a child that starts a transaction. The response to the web service means “I’ve received your request”, NOT “I’ve completed your request”.
- The parent transaction continues executing business logic.
- At some later point in time the parent transaction may issue another synchronous web service call to retrieve the results.

Another way to think about it is that if there is a race condition between the parent and the child transaction such that it is unknown whether the child transaction will complete before the parent transaction continues executing, it is appropriate to use the Asynchronous Flow flag. The circumstance in which it is inappropriate to use the Asynchronous Flow flag is when the caller makes a synchronous call using a synchronous protocol and blocks waiting for the child transaction to complete. It would be appropriate to use the Asynchronous Flow flag if a message is sent using a message queuing protocol even if the application immediately blocks to await a response.

When the flag is FALSE, the ARM agent cannot presume anything about the nature of parent-child interaction.

When used, this flag needs to be set prior to specification of a parent correlator on ArmTransaction **start()**, ArmTranReport **setParentCorrelator()**, or ArmTranReport **generateCorrelator()**. The flag could be set by the child recipient of the correlator, but the expectation is that the flag is set most frequently by the parent prior to the control transfer

protocol invocation. Note that the Independent Transaction (IT) flag, described below, has a semantic dependency on this flag, which also potentially influences where this flag is set.

The Asynchronous Flow flag is interpreted only within the scope of a single parent-child control transfer, and is exclusively manipulated by applications. The ARM agent never sets this flag. The application uses ArmCorrelator **setAsynchronous()** to set the flag. All correlators constructed by the ARM agent and returned as output on ArmTransaction **getCorrelatort()** or ArmTranReport **generateCorrelator()** are constructed with the Asynchronous Flow flag set to FALSE. Note that the flag is never inherited from parent correlators to child correlators, unlike the Agent Trace and Application Trace flags, which often are inherited from the parent correlator.

4.2.2 Indicating Independent Flows

The Independent Transaction flag indicates that the invoked transaction will not influence the parent transaction in any way, and in many cases, the invoked transaction would be best thought of and modeled as a new business transaction.

Setting the Independent Transaction flag to TRUE indicates that ARM-reported program logic execution in the child recipient of this correlator reflects transaction work that, although initiated by this parent, is performed independently of the parent's ARM-reported transaction work in scope and purpose. More specifically, setting the flag to TRUE indicates that neither the parent transaction's response time nor status is dependent on the child transaction's execution, nor is there any expected use of the same transaction context. It would be typical for this new independent transaction to be classified for purposes such as a service class independent of the parent's classification.

It logically follows that the child transaction is executing asynchronously to the parent transaction and therefore the Independent Transaction flag may be set to TRUE only if the Asynchronous Flow flag is also set to TRUE. If the Asynchronous Flow flag is FALSE, the Independent Transaction flag is ignored.

Asynchronous Flow Flag	Independent Transaction Flag	Interpretation
FALSE	FALSE	No assumptions about the parent/child interaction can be made.
FALSE	TRUE	The Independent Transaction flag is ignored. No assumptions about the parent/child interaction can be made.
TRUE	FALSE	The child transaction is executing asynchronously to the parent transaction but not independently.
TRUE	TRUE	The child transaction is executing asynchronously to the parent transaction and is also independent of the parent transaction.

Table 4: Correlator Flags in Asynchronous Flows

When used, this flag needs to be set prior to specification of a parent correlator on ArmTransaction **start()**, ArmTranReport **setParentCorrelator()**, or ArmTranReport **generateCorrelator()**. The flag could be set by the child recipient of the correlator, but the

expectation is that the flag is set most frequently by the parent prior to the control transfer protocol invocation.

The Asynchronous Flow and Independent Transaction flags are interpreted only within the scope of a single parent-child control transfer, and are exclusively manipulated by applications. The ARM agent never sets these flags. The application uses ArmCorrelator **setAsynchronous()** and ArmCorrelator **setIndependentTran()** to set the flags. All correlators constructed by the ARM agent and returned as output on ArmTransaction **getCorrelatort()** or ArmTranReport **generateCorrelator()** are constructed with the Asynchronous Flow and Independent Transaction flags set to FALSE. Note that the flags are never inherited from parent correlators to child correlators, unlike the Agent Trace and Application Trace flags, which often are inherited from the parent correlator.

4.2.3 Event Flows (ARM 4.1)

Asynchronous flows between programs are initiated, controlled, and terminated through the exchange of messages. It can be useful to programs that analyze asynchronous flows and how they impact applications to understand the underlying event flows.

There are two types of events:

- Message Received

  The Message Received Event sub-buffer is optionally used to describe one or more messages that have been received that have an effect on the execution state of a transaction. The cases of interest are the following:

  — A message causes the transaction to initiate.

  — A message causes the transaction to unblock.

  — A message is received that terminates an asynchronous transaction or a step in an asynchronous transaction.

- Message Sent

  The Message Sent Event sub-buffer is optionally used to describe one or more messages that have been sent that have an effect on the execution state of a transaction. The cases of interest are the following:

  — A message is sent that causes the transaction to block.

  — A message is sent that initiates or terminates an asynchronous transaction or a step in an asynchronous transaction.

  — A message is sent that is part of an exchange between this transaction and another transaction.

With either a Message Received or Message Sent event an application can indicate with the End-Of-Flow flag that the current transaction is the last in a series of transactions that together represent one logical flow. For example, if there are three transactions (A,B,C) that together perform some service, and if the flow between them is A invokes B, which invokes C, and

neither B nor C return status or other data to the transaction that invoked it, then C represents the end of the flow A→B→C.

The event data is provided by storing ArmMessageReceivedEvent and ArmMessageSentEvent in an array in ArmMessageEventGroup, and then associating the messages with an ArmTransaction using ArmTransaction **setMessageEventGroup()**.

5 Describing Applications and Transactions

ARM 4.1 uses two types of properties to describe applications and transactions: "identity" properties (for applications and transactions), and "context" properties (for applications and transactions). Each property consists of a name string and a value string.

They differ based on when the names and/or values are set, as shown in Table 5.

Type of Property	Same for all Transactions of a Given Type	May Vary per each Time a Transaction Executes
Identity Property (applications and transactions)	Name Value	
Context Property (applications and transactions)	Name	Value

Table 5: Descriptive Property Types

When deciding which property type to use, instrumenters should be aware of the trade-offs:

- Processing of identity properties can generally be optimized more than processing of context properties because it can be done once at registration time for both the names and the values and apply to all transactions of the registered type.
- Processing of context property values may occur for every executing transaction. This increases overhead but it is a good practice if each transaction "flavor", where a flavor represents a different combination of properties, is a slight variation on the same type of transaction. An implementation that does not process the context properties could still provide useful reports if the performance characteristics of each flavor are similar.

5.1 Identity Properties

An identity property is a property that has the same name and value for all instances of an application or transaction.

- The *application name* and *transaction name* parameters are mandatory and provide the most basic identification. Many applications and transactions are satisfactorily identified through the use of only the *name* parameter.
- In addition, there can be up to 20 identity properties of the pattern (name,value) pair, in which both name and value are separate character strings.
- Transactions can also have a URI property that is part of the identity when it is provided.

Identity property names and values are provided when an application or transaction is registered.

5.2 Context Properties

A context property name is the same for all instances of an application or transaction, but each instance may have different values.

- In addition, applications have two implicitly named context properties – the group name and instance name.
- In addition, transactions can also have a URI property and a user name property that can be used to help define the context.

Context property names are provided when an application or transaction is registered.

Context property values are provided when an application or transaction starts executing.

6 Transaction Response Time Elements

The use of ARM implies a model in which the total response time of a transaction can be sub-divided into finer grained elements, as shown in Figure 15:.

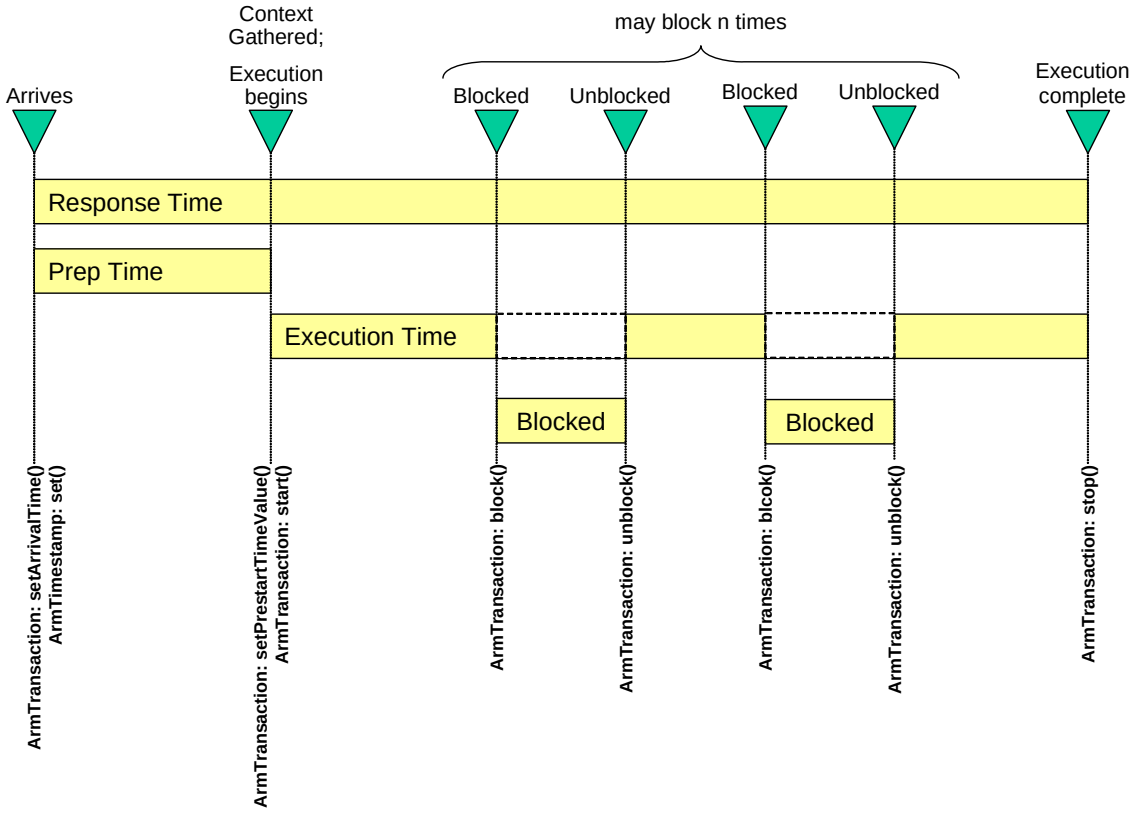

Figure 15: Response Time Elements

- The processing begins when the transaction “arrives”; i.e., when the message (including synchronous RPC invocations) that invokes the transaction is retrieved by the application. The message could have been delayed substantially prior to retrieval, such as a message sitting in a queue.
- In some environments the application first gathers context related to the transaction before it can be processed and/or ArmTransaction **start()** called. For example, instrumentation in some applications makes a JNDI call to retrieve some context properties passed on ArmTransaction **start()**. This is shown in the example as Preparation Time. However, because this Prep Time is part of the response time, the application captures a timestamp as soon as it starts executing. It will later provide the timestamp as a parameter on ArmTransaction **start()**.

- The transaction begins executing. The application calls ArmTransaction **start()** to indicate the fact. For environments that do not have any Prep Time, the actual time that ArmTransaction **start()** is called is used as the time that the response time measurement starts.
- The transaction may block one or more times waiting for an external event, such as a database call or a call to another application program. It indicates the beginning and end of each period when it is blocked with ArmTransaction **blocked()** and **unblocked()**, respectively.
- The transaction completes executing and indicates the same to ARM with ArmTransaction **stop()**. The actual time that ArmTransaction **stop()** is called is used as the time that the response time measurement ends.

6.1 Arrival and Preparation Time

ARM 4.0 specified one way [ArmTransaction **setArrivalTime()**] to indicate the duration of the Prep Time. With ARM 4.1, there are three ways to indicate the duration of the Prep Time.

6.1.1 Opaque Timestamp (ARM 4.0/ARM 4.1)

Capture a timestamp when the transaction arrives and processing begins using either ArmTransaction **setArrivalTime()** or the ArmTimestamp subclass ArmTimestampOpaque along with ArmTransaction **setPrestartTimeValue()**, as shown in the example in Figure 15:. The two approaches are equivalent. Using ArmTimestampOpaque is preferred starting with ARM 4.1.

6.1.2 Formatted Timestamp (ARM 4.1)

Capture the arrival time using some unspecified means, convert it into one of the formats that ARM recognizes (ArmTimestampStrings or ArmTimestampUsecJan1970), and provide it with ArmTransaction **setPrestartTimeValue()**.

6.1.3 Measured Prep Time (ARM 4.1)

Measure the Prep Time (or a mean over several transaction executions) as a duration, store it in a long integer or ArmPrestartTimeStats, and provide it with ArmTransaction **setPrestartTimeValue()**.

6.2 Blocked Time

A blocking condition is one in which an application suspends execution of a transaction while awaiting the completion of an external event, such as the completion of a database query or other service request. If an application makes a service request but is able to continue executing other logic, it is not in a blocked condition. After completing the other logic it may enter a blocked condition if it is still waiting for the previous service request to complete before continuing.

The application indicates that it is blocked and unblocked using ArmTransaction **blocked()** and **unblocked()**. The moment at which either call is made is considered the time when the blocked condition begins or ends.

Beginning with ARM 4.1, the application may optionally provide information indicating whether the blocking event is a synchronous or an asynchronous event, plus some additional information describing the cause of the blocking condition. This data is passed in ArmBlockCause with ArmTransaction **blocked()**.

6.3 Thread Binding

Independent of blocking conditions, it can be useful to know which threads are executing which transactions. The thread binding could be useful for managing computing resources at a finer level of granularity than a process.

This information is not readily apparent to the operating system because it does not inspect the context of each thread and does not know the association of a thread to a transaction measured with ARM. The application does know the binding and can indicate the same to ARM. There are two ways to indicate the binding:

- ArmTransaction **bindThread()** and **unbindThread()**. **bindThread()** indicates that the thread from which it is called is performing on behalf of the ArmTransaction. A transaction remains bound to a thread until any of ArmTransaction **unbindThread()**, **stop()**, or **reset()** is called.
- ArmTransaction **setAutomaticBindThread(true)** is called. This causes this ArmTransaction to be bound to the current thread every time **start()** is called until the **setAutomaticBindThread(false)** is called. Setting this flag is equivalent to immediately calling ArmTransaction **bindThread()** after **start()** completes, except the timing is a little more accurate and the extra call is avoided.

There can be any number of threads simultaneously bound to the same transaction.

Note that **bindThread()** and **blocked()** are used independently of each other.

7 Additional Data about a Transaction

The identification information 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 whether the performance was good. Additional information – such as the number of bytes or files backed-up – provides 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.
- It can be useful to know additional context information about a transaction, especially if it fails. For example, instrumentation about a servlet might provide the following diagnostic properties when there is a failure: query string, remote host, remote address, remote user, protocol, request attributes, request header, request parameters, etc.

ARM provides four ways for applications to provide these types of data. The use of either is *optional*.

- One is the identity and context properties described in Chapter 5.
- One is a set of short numeric or character strings named "metrics" in ARM. They are described in Section 7.1.
- One is a long character string, named "diagnostic detail". It is described in Section 7.2.

- One is a set of properties, named “diagnostic properties”. They are described in Section 7.3.

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.

7.1 Metrics

7.1.1 Metric Data Types

Metrics are provided for executing transactions.

- Each metric has a format and name. These are provided when a transaction is registered and do not change afterwards.
- The metric values may vary each time the transaction executes.
- The metric values are provided at any or all of when a transaction starts, stops, or is updated between the start and stop.

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

7.1.1.1 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) RFC 1155 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).

7.1.1.2 Gauges

A gauge value can go up and down, and it can be positive or negative. This is the IETF RFC 1155 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 the amount of memory used over 20 transactions in an interval is measured and the average usage for each of these transactions was

15MB, it does not make sense to say that 20*15=300MB of memory were used over the interval. It would make sense to say that the average was 15MB, that the median was 12MB, and that the standard deviation was 8MB. 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).

7.1.1.3 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

7.1.1.4 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 one string type:

- Strings of 1-32 characters: ArmMetricString32

7.1.2 How to Provide Metrics

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

7.1.2.1 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. Each ArmMetric subclass supports

the **set()** method. Figure 16 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.

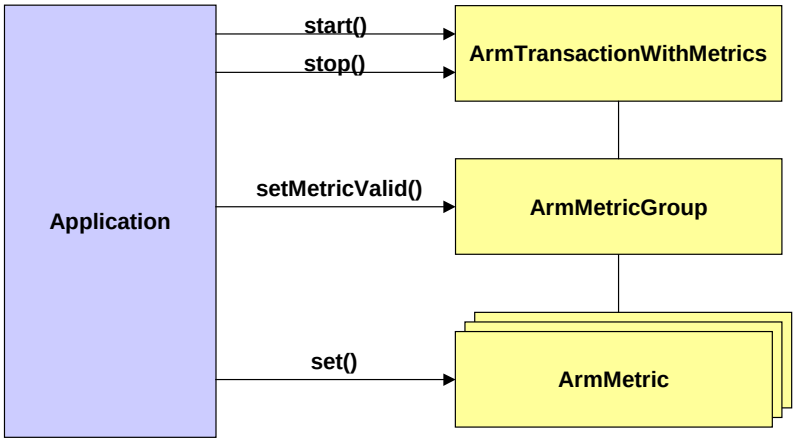

Figure 16: Providing Additional Data Using ArmTransaction and ArmMetric

Prior to calling **start()**, **update()**, or **stop()**, the application may set the value in each metric. The ArmMetricGroup method **setMetricValid()** is used to indicate whether the data is valid. This is needed because the data might be valid only when **stop()** is executed, as an example. Figure 16 shows this process.

7.1.2.2 Using ArmTranReportWithMetrics

The process is similar if the application is calling ArmTranReport **report()**. It creates instances of subclasses of ArmMetric (e.g., ArmMetricCounter32) and binds an instance to an ArmTranReportWithMetrics instance using ArmMetricGroup. Each ArmMetric subclass supports the **set()** method. Figure 17 shows this process. ArmTranReportWithMetrics is a subclass of ArmTranReport and, hence, implements all the methods of ArmTranReport, in addition to some methods for manipulating metrics.

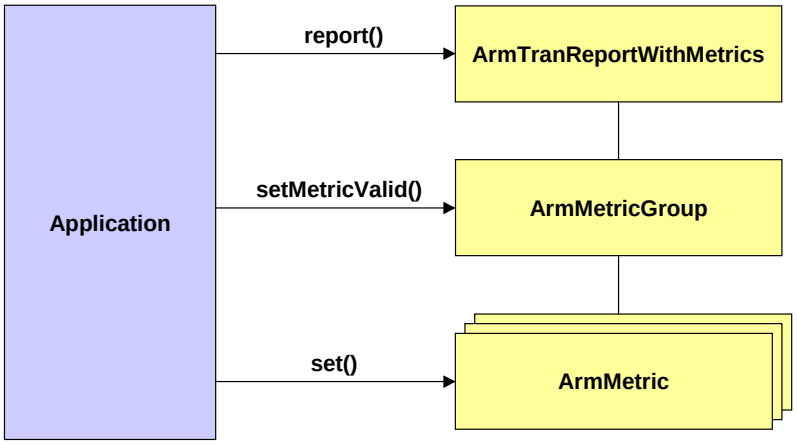

Figure 17: Providing Additional Data using ArmTranReport

Prior to calling **report()**, the application may set the value in each metric. The ArmMetricGroup method **setMetricValid()** is used to indicate whether the data is valid. This is needed because the data may not always be valid. Figure 17 shows this process.

7.1.3 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.

7.1.3.1 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 transactions, so the value often changes from one **stop()** to the next **start()** for the same transaction object.

7.1.3.2 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.

7.1.3.3 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. 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. This convention is identical to the string convention.

7.1.3.4 Strings

The last value passed when any of the **start()**, **update()**, or **stop()** calls are made will be the value attributed to the transaction. 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. This convention is identical to the numeric ID convention.

7.2 Diagnostic Detail

Diagnostic detail may be provided for executing transactions.

- The diagnostic detail is in the form of a long null-terminated character string.
- The diagnostic detail string is provided when a transaction stops [ArmTransaction **stop()**].
- There are no constraints on the contents of the string. It can contain any information that the application thinks may be useful. For example, if a database query fails, the application might provide the text of the SQL query.

The application may provide either Diagnostic Detail OR Diagnostic Properties for any transaction, but not both.

7.3 Diagnostic Properties (ARM 4.1)

Diagnostic properties may be provided for executing transactions.

- Each property consists of a name and a value, both strings.
- Both names and values are provided when a transaction stops [ArmTransaction **stop()**].
- Both names and values may vary each time a transaction executes. They are particularly useful in some middleware environments in which the property names used by hosted applications are not known when the middleware starts executing.

The application may provide either Diagnostic Detail OR Diagnostic Properties for any transaction, but not both.

8 Creating ARM Objects

8.1 Overview of Java Interfaces

This document 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.arm40.metric package, and it defines four method signatures: **getDefinition()**, **getMetric(**int**)**, **isMetricValid(**int**)**, and **setMetricValid(**int,boolean**)**.

```
package org.opengroup.arm40.metric;
public interface ArmMetricGroup
{
    public ArmMetricGroupDefinition getDefinition();
    public ArmMetric getMetric(int index);
    public boolean isMetricValid(int index);
    public int setMetricValid(int index, boolean value);
}
```

A program cannot create an instance (object) of an interface because there is 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.arm40.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.arm40.metric.*;
public class MyGroup
    implements ArmMetricGroup
{
    public ArmMetricGroupDefinition getDefinition();
    { // program code goes here }
    public ArmMetric getMetric(int index);
    { // program code goes here }
    public boolean isMetricValid(int index);
    { // program code goes here }
    public int setMetricValid(int index, boolean value);
    { // program code goes here }
    public int privateStuff();
    { // program code goes here }
}

import org.opengroup.arm40.metric.*;
public class AnotherVendorsOne
```

```
    implements ArmMetricGroup
{
    public ArmMetricGroupDefinition getDefinition();
    { // program code goes here }
    public ArmMetric getMetric(int index);
    { // program code goes here }
    public boolean isMetricValid(int index);
    { // program code goes here }
    public int setMetricValid(int index, boolean value);
    { // 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 five methods of MyGroup. g5 is of type ArmMetricGroup so it can execute only the four 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.setMetricValid(mySlot, true);
group.setMetricValid(mySlot, false);
group.privateStuff();
```

8.2 Creating ARM Objects in an Application

The discussion so far 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 4.0 Java Bindings 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 ARM specification. 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. This document defines three 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 three factory interfaces are ArmMetricFactory, ArmTranReportFactory, and 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 (Java Virtual Machine). It provides a single place to store property values that will be available to all programs running within the JVM.

ARM defines three property keys, one for each of the three 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 ArmMetricFactory interface assigns the value ArmMetricFactory to its constant variable propertyKey. Each of the following statements is in the respective factory interfaces:

```
public static final String propertyKey = "Arm40.ArmMetricFactory";
public static final String propertyKey = "Arm40.ArmTranReportFactory";
public static final String propertyKey = "Arm40.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 two 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 three 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 and related classes. Three are created in this example, all for the same transaction ID. In this example, the **Class.forName()** method specifically names the system class loader, which can avoid problems if multiple class loaders are involved and the factory implementation uses JNI (Java Native Interface).

```
// Create the factory object
Properties p = System.getProperties();
String keyTranFactoryClass = ArmTransactionFactory.propertyKey;
String tranFactoryName = p.getProperty(keyTranFactoryClass);
Class  tranFactoryClass = Class.forName(tranFactoryName, true,
    ClassLoader.getSystemClassLoader());
ArmTransactionFactory tranFactory;
tranFactory = (ArmTransactionFactory) tranFactoryClass.newInstance();

// The application and transaction objects
ArmApplicationDefinition myAppDefn;
ArmApplication myApp;
ArmTransactionDefinition myTranDefn;
ArmTransaction tran1, tran2, tran3;

// Create application objects
myAppDefn = tranFactory.newArmApplicationDefinition("My Application",
    null, null);
myApp = tranFactory.newArmApplication(myAppDefn, null, null, null);

// Create three transaction objects
myTranDefn = tranFactory.newArmTransactionDefinition(
    myAppDefn, "My Transaction", null, null);
tran1 = tranFactory.newArmTransaction(myApp, myTranDefn);
tran2 = tranFactory.newArmTransaction(myApp, myTranDefn);
tran3 = tranFactory.newArmTransaction(myApp, myTranDefn);
```

8.2.1 Version Compatibility

ARM 4.1 extends ARM 4.0 by adding methods to existing interfaces and adding new interfaces. All these changes are in the same packages defined in ARM 4.0:

- org.opengroup.arm40.metric
- org.opengroup.arm40.tranreport
- org.opengroup.arm40.transaction

This creates the possibility that the application that calls the ARM interfaces and the implementation of the ARM interfaces may be at different levels. There are two cases to consider:

- An application using ARM 4.0 may use an ARM 4.1 implementation. In this case there are no compatibility issues because an ARM 4.1 implementation must implement all of the ARM 4.0 interfaces.
- An application using ARM 4.1 may instantiate classes that implement ARM 4.0 but not ARM 4.1. In this case there are compatibility issues because the application may use interfaces and/or methods that are not implemented. The rest of this section describes how an application avoids runtime errors in this scenario.

Each of the packages has one factory with the following names:

- ArmMetricFactory
- ArmTranReportFactory
- ArmTransactionFactory

In ARM 4.0 each factory interface defines one static property (propertyKey), and its use is described in Section 8.2. In ARM 4.1 an additional static property (propertyKey41) is defined. It is used similarly. An implementation that implements only ARM 4.0 will set a class name value in propertyKey but not in propertyKey41. An implementation that implements ARM 4.1 will set a class name value in both propertyKey and propertyKey41.

- An application that uses ARM 4.0 will use propertyKey to look up the factory class name in java.util.Properties.
- An application that uses ARM 4.1 will use propertyKey41 to look up the factory class name in java.util.Properties. If propertyKey41 is null, then the implementation is ARM 4.0 but not ARM 4.1. In this case, the application must avoid calling any method or interface that was added in ARM 4.1. If it does call such a method, the likely result is a runtime error because there will not be a concrete class available that implements the method. The application has the option to instantiate the factory that implements ARM 4.0 and restrict its use to the ARM 4.0 capabilities.

8.3 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 code base. By default, the code base is the directory that contains the HTML file that loaded the applet. The HTML file can specify the code base 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 code base. For classes that are members of a package, the code base is extended by the package name. For example, if the code base is www.abc.com/test, the ARM package would be in directory www.abc.com/test/org/opengroup/arm40/transaction. An applet can get the URL of the code base 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 code base 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 three 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()+ System.getProperty("file.separator")
    + "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();

// The application and transaction objects
ArmApplicationDefinition myAppDefn;
ArmApplication myApp;
ArmTransactionDefinition myTranDefn;
ArmTransaction tran1, tran2, tran3;

// Create application objects
myAppDefn = tranFactory.newArmApplicationDefinition("My Application",
    null, null);
myApp = tranFactory.newArmApplication(myAppDefn, null, null, null);

// Create transaction objects
myTranDefn = tranFactory.newArmTransactionDefinition(myAppDefn,
    "My Transaction", null, null);
```

```
tran1 = tranFactory.newArmTransaction(myApp, myTranDefn);
```

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

9 Error Handling Philosophy

The error handling philosophy of the ARM specification 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.

9.1 Errors for which the Application Should Test

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, which 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()**.

9.2 Errors for which the Application Does Not Need to Test

None of the interfaces in the three packages that comprise the ARM specification 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 and test for them, but has the opportunity to do so. To enable this, an application can test method return codes, get an error status from an object, and register a callback that is invoked when an error occurs.

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.

Except for the callback via the ArmErrorCallback interface, the content, format, and delivery mechanisms for error notifications are not part of the ARM specification. They are implementation-defined. A good implementation will provide sufficient detail to not only detect that a problem occurred, but to also isolate and resolve the problem.

9.3 How an Application Tests for Errors

If an application chooses to test for errors that don't result in throwing an exception, it may do so in three ways. Note again that testing for these errors is entirely optional. They can be ignored (the data about the transactions and related entities may not be valid or useful to an ARM implementation, but the application's execution will not be disrupted in any way).

Anytime a method results in an error the ARM implementation does the following:

- Sets the object's error code. This error code can be retrieved with the **getErrorCode()** method of ArmInterface, from which all other ARM interfaces with methods derive.
- Calls **errorCodeSet()** in ArmErrorCallback, if an ArmErrorCallback has been successfully registered with the factory that was used to create the object.
- Returns the error code if the method that caused the error returns an error code.

Anytime a method of an object is invoked, any previous error code value is overwritten with the status of the last method invoked.

9.3.1 Testing a Method Return Code

Many methods return an int. Unless otherwise noted, the returned integer is an error code. It has a value of zero if no error occurred. Any non-zero value indicates that there was an error.

9.3.2 Testing an Object Error Code

Many methods may result in the ARM implementation detecting an error. Some of these methods return an error code and some do not. Regardless of whether a method returns an error code, all methods that detect an error set the object's error code. The application may retrieve the error code with the **getErrorCode()** method of ArmInterface, from which all other ARM interfaces with methods derive. The application may use **getErrorMessage()** to get a string message describing the error.

9.3.3 Registering a Callback

An application may create an object that implements the ArmErrorCallback interface and register it with the **setErrorCallback()** method of one or all of the factory objects that are used to create objects on behalf of the application. **setErrorCallback()** returns a boolean indicating whether the registration was accepted. If it was accepted, any method that causes an object's error code to be set to a non-zero value will cause the ArmErrorCallback method **errorCodeSet()** to be invoked, passing a reference to the object and the name of the interface and method in which the error was detected.

10 Instrumentation Control (ARM 4.1)

ARM is designed to be a high performing interface so applications and middleware can invoke it as often as there is an interesting event to report. The ARM implementation determines whether to process the passed data and the manner of processing. In most cases this is entirely satisfactory and the application does not include conditional logic to determine when to call ARM nor how much data to provide with each call.

Certain applications or middleware may provide fine-grained instrumentation, both in terms of the number of instrumentation points and the depth of data available at each instrumentation point. IBM WebSphere is an example. Beginning with ARM 4.1, ARM provides a mechanism that enables an application or middleware program to query the ARM implementation for the desired instrumentation granularity. The mechanism works as shown in Figure 18:.

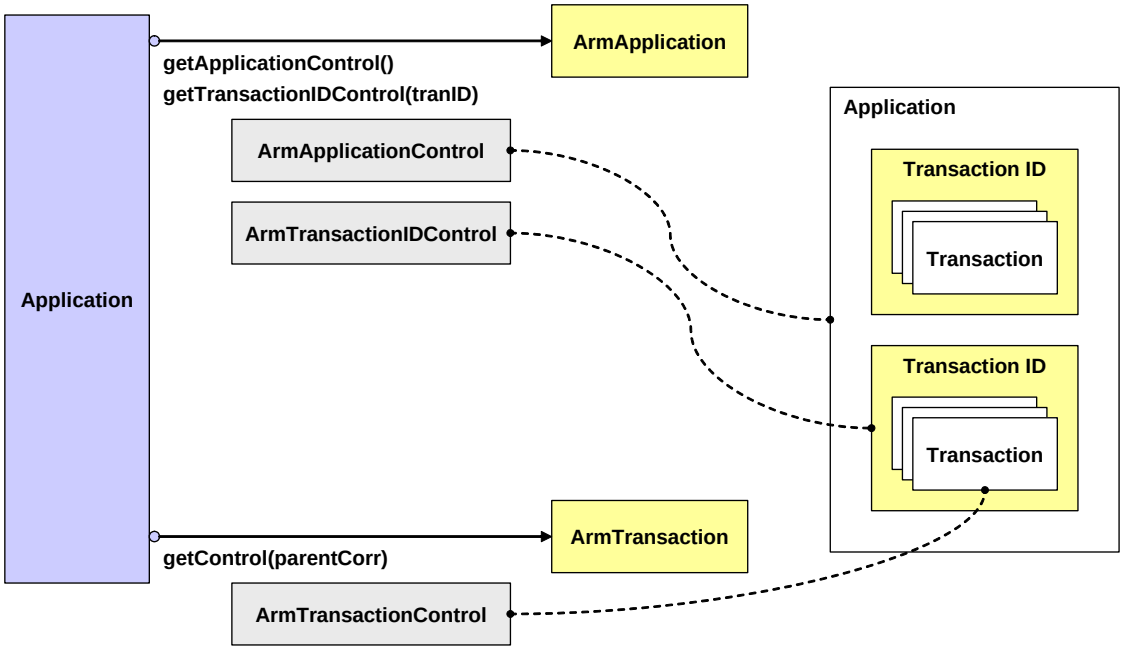

Figure 18: Instrumentation Control Interfaces

The use of instrumentation control is optional for both instrumented applications and ARM implementations, and there is a handshake so an application knows whether to depend on the capability. An application determines whether instrumentation control is being used, and if so, establishes the instrumentation level for the entire application, by calling ArmApplication **getApplicationControl()**, then calling ArmApplicationControl's methods to determine the control settings. If **getApplicationControl()** returns null, the ARM implementation is not using instrumentation control and the application uses its default instrumentation settings.

Two methods in ArmApplicationControl indicate whether instrumentation control is being used at the Transaction ID and/or Transaction scopes, respectively:

- **isTransactionDefinitionControlUsed()**
- **isTransactionControlUsed()**

There are three control scopes. Table 6 summarizes the available controls for each scope.

- Application-wide for all transactions (ArmApplicationControl). An application determines whether instrumentation control is being used, and if so, establishes the instrumentation level for the entire application, unless overridden by ArmTransactionDefinitionControl or ArmTransactionControl scope.
- Application-wide for all transactions of a specified ID (ArmTransactionDefinitionControl). If **isTransactionDefinitionControlUsed()** true, then the application is invited to request the instrumentation controls once and then they will apply for every transaction with a registered transaction ID, unless overridden by ArmTransactionControl.
- Individual executing transaction (ArmTransactionControl). If the ARM implementation returned **isTransactionControlUsed()**=true, then the application is invited to request the instrumentation controls for each executing transaction.

Control Name	Brief Description	ArmApplicationControl	ArmTransactionDefinitionControl	ArmTransactionControl
isTransactionDefinitionControlUsed()	Indicates whether the ARM implementation uses ArmTransactionDefinitionControl.	X		
isTransactionControlUsed()	Indicates whether the ARM implementation uses ArmTransactionControl.	X		
isPrivateDataRequested()	Indicates whether private data (e.g., account numbers) should be provided in any form, such as in a metric or diagnostic data, or in a correlator. The application determines what constitutes private data.	X		

Control Name	Brief Description	ArmApplicationControl	ArmTransactionDefinitionControl	ArmTransactionControl
isSecureDataRequested()	Indicates whether secure data (e.g., passwords) should be provided in any form, such as in a metric or diagnostic data, or in a correlator. The application determines what constitutes secure data.	X		
isBindThreadRequested()	Indicates whether the implementation prefers that the application uses ArmTransaction's **bindThread()** and **unbindThread()** calls.	X	X	X
isBlockRequested()	Indicates whether the implementation prefers that the application uses ArmTransaction's **blocked()** and **unblocked()** calls.	X	X	X
isDiagnosticDataRequested()	Indicates whether the implementation prefers that the application uses ArmDiagnosticProperties or the diagnostic detail string on ArmTransaction's **stop()** call.	X	X	X
isMessageEventDataRequested()	Indicates whether the implementation prefers that the application inform it about message exchanges and other asynchronous flows.	X	X	X
isMetricDataRequested()	Indicates whether the implementation prefers that the application provides ArmMetric data.	X	X	X
isUserDataRequested()	Indicates whether the implementation prefers that the application uses ArmTransaction's **setUser()** call.	X	X	X

Table 6: Instrumentation Control Scope

11 The ARM 4.0 Data Model

11.1 Data Model Using ArmTransaction

Figure 19 shows the data model when using ArmTransaction. To avoid clutter in the diagrams, some interfaces – such as ArmCorrelator, ArmID, and ArmUser – are not shown as separate classes.

Four interfaces must be used: ArmApplicationDefinition, ArmTransactionDefinition, ArmApplication, and ArmTransaction. All others are optional. The relationships between interfaces are implemented via Java object references.

- ArmApplicationDefinition describes metadata about the application. It is the root object that anchors all other objects. There is one per application. Note, however, that an ARM application is a logical entity. The same process may support several very different logical applications.
- ArmTransactionDefinition describes metadata about a type of transaction. There may be any number of transaction definitions in each application.
- ArmIdentityProperties and its subclass ArmIdentityPropertiesTransaction are used to provide additional optional metadata about applications and transactions.
- ArmApplication represents a running application. It is common for there to be one running instance per definition, but there could be any number of running instances per definition.
- ArmTransaction represents an executing transaction. It is the most important and the most widely used ARM interface. More specifically, the application uses the paired methods **start()** and **stop()** to indicate the beginning and end of the transaction. At all other times the ArmTransaction object is unused and does not represent anything. ArmTransaction objects may be re-used. An application may create one ArmTransaction per thread, or it might create a pool of ArmTransaction instances, and re-use them on an as-needed basis.

  The measurements common to all transactions are status, response time, and the time-of-day when the transaction executed. Optionally, the application can also provide data to correlate parent and child transactions and context data such as the user on whose behalf the transaction executes. Substitute ArmTransactionWithMetrics if additional metric data is provided.
- 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 with 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 is used to bind a set of ArmMetric objects to an ArmTransactionWithMetrics object.
- ArmTransactionWithMetricsDefinition, ArmMetricGroupDefinition, and ArmMetricDefinition are used to provide metadata about metric groups and metrics.

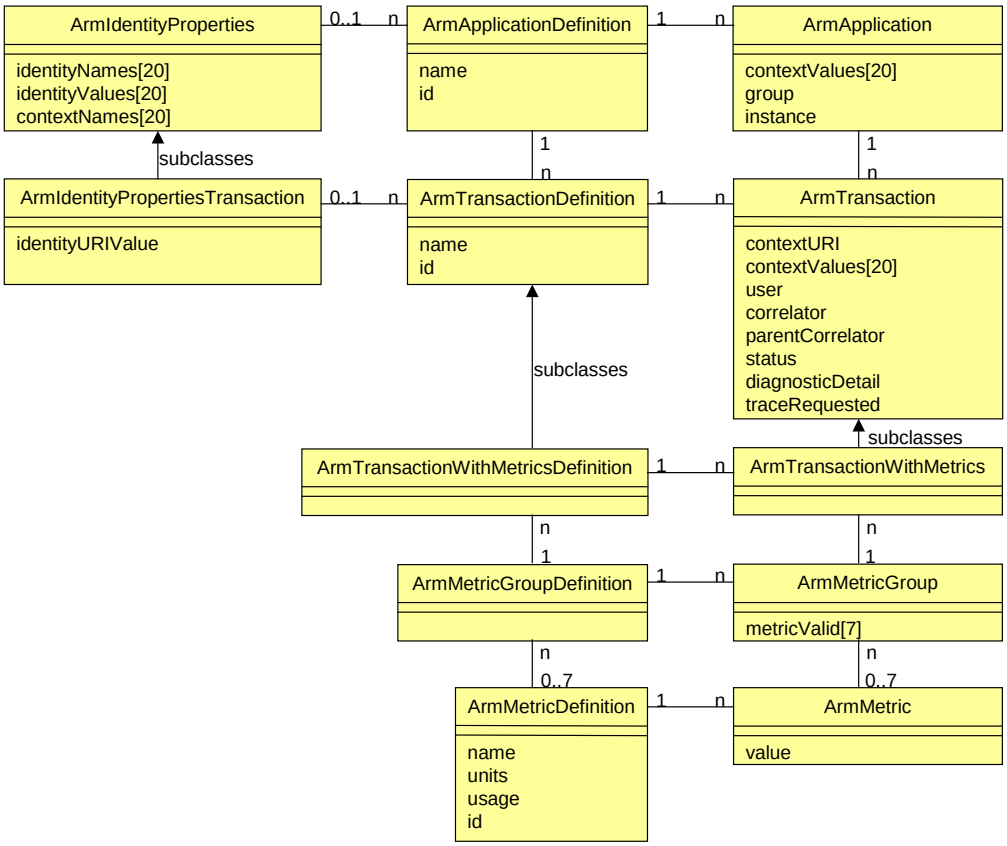

Figure 19: ARM 4.0 Data Model Using ArmTransaction

11.2 Data Model Using ArmTranReport

Figure 20 summarizes the data model when using ArmTranReport. All the interfaces that are already depicted in Figure 19 and described above are not shaded. Only the interfaces that specifically support the use of ArmTranReport are shared and described below.

- ArmTranReport and its subclass ArmTranReportWithMetrics 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 objects, populates it, then calls **report()**. As soon as **report()** returns, the application can re-use the ArmTranReport instance.

- ArmApplicationRemote may execute on the local system or a remote system. When it executes on the local system, it is associated with an ArmApplication object. When it executes on a remote system, it is associated with a subclass of ArmApplication – ArmApplicationRemote. ArmApplicationRemote differs from ArmApplication only in that it may be associated with an ArmSystemAddress object. An ArmSystemAddress object contains the network address of the remote system in one of several possible formats.

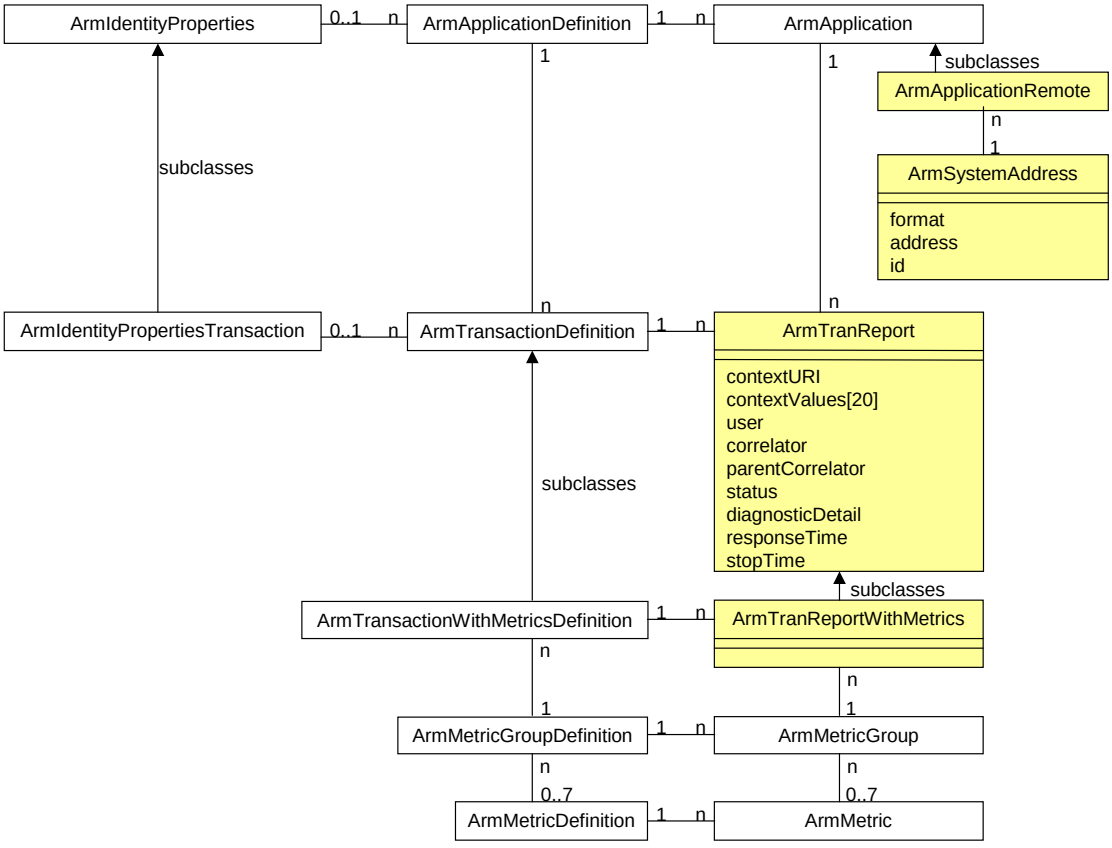

Figure 20: ARM 4.0 Data Model Using ArmTranReport

12 The org.opengroup.arm40.* Packages

12.1 Interface List (by Java Package)

The following table lists all the interfaces, arranged by package name, and in 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.arm40. transaction	org.opengroup.arm40. tranreport	org.opengroup.arm40. metric
ArmApplication	ArmApplicationRemote	ArmMetric
ArmApplicationControl	ArmSystemAddress	ArmMetricCounter32
ArmApplicationDefinition	ArmTranReport	ArmMetricCounter32Definition
ArmBlockCause	ArmTranReportFactory	ArmMetricCounter64
ArmConstants		ArmMetricCounter64Definition
ArmCorrelator		ArmMetricCounterFloat32
ArmDiagnosticProperties		ArmMetricCounterFloat32Definition
ArmErrorCallback		ArmMetricDefinition
ArmID		ArmMetricFactory
ArmIdentityProperties		ArmMetricGauge32
ArmIdentityPropertiesTransaction		ArmMetricGauge32Definition
ArmInterface		ArmMetricGauge64
ArmMessageEvent		ArmMetricGauge64Definition
ArmMessageEventGroup		ArmMetricGaugeFloat32
ArmMessageReceivedEvent		ArmMetricGaugeFloat32Definition
ArmMessageSentEvent		ArmMetricGroup
ArmPrestartTimeStats		ArmMetricGroupDefinition
ArmTimestamp		ArmMetricNumericId32
ArmTimestampOpaque		ArmMetricNumericId32Definition
ArmTimestampStrings		ArmMetricNumericId64
ArmTimestampUsecJan1970		ArmMetricNumericId64Definition
ArmToken		ArmMetricString32
ArmTransaction		ArmMetricString32Definition
ArmTransactionControl		ArmTranReportWithMetrics
ArmTransactionDefinition		ArmTransactionWithMetrics
ArmTransactionFactory		ArmTransactionWithMetricsDefinition
ArmTransactionDefinitionControl		

org.opengroup.arm40. transaction	org.opengroup.arm40. tranreport	org.opengroup.arm40. metric
ArmUser		

12.2 Interface List (in Alphabetical Order)

The following table lists in alphabetical order all the interfaces that comprise the ARM specification, and lists the Java package in which they can be found. It also lists whether instances of the interface can be instantiated. Some interfaces cannot be instantiated because they define only constants or they are abstract superclasses. To create instances of instantiable interfaces, use the appropriate method of the factory class for the package.

Class Name (in Alphabetic Order)	Java Package	Can be Instantiated?
ArmApplication	org.opengroup.arm40.transaction	Yes
ArmApplicationControl	org.opengroup.arm40.transaction	Yes
ArmApplicationDefinition	org.opengroup.arm40.transaction	Yes
ArmApplicationRemote	org.opengroup.arm40.tranreport	Yes
ArmBlockCause	org.opengroup.arm40.transaction	Yes
ArmConstants	org.opengroup.arm40.transaction	No
ArmCorrelator	org.opengroup.arm40.transaction	Yes
ArmDiagnosticProperties	org.opengroup.arm40.transaction	Yes
ArmErrorCallback	org.opengroup.arm40.transaction	Yes
ArmID	org.opengroup.arm40.transaction	Yes
ArmIdentityProperties	org.opengroup.arm40.transaction	Yes
ArmIdentityPropertiesTransaction	org.opengroup.arm40.transaction	Yes
ArmInterface	org.opengroup.arm40.transaction	No
ArmMessageEvent	org.opengroup.arm40.transaction	No
ArmMessageEventGroup	org.opengroup.arm40.transaction	Yes
ArmMessageReceivedEvent	org.opengroup.arm40.transaction	Yes
ArmMessageSentEvent	org.opengroup.arm40.transaction	Yes
ArmMetric	org.opengroup.arm40.metric	No
ArmMetricCounter32	org.opengroup.arm40.metric	Yes
ArmMetricCounter32Definition	org.opengroup.arm40.metric	Yes
ArmMetricCounter64	org.opengroup.arm40.metric	Yes
ArmMetricCounter64Definition	org.opengroup.arm40.metric	Yes
ArmMetricCounterFloat32	org.opengroup.arm40.metric	Yes
ArmMetricCounterFloat32Definition	org.opengroup.arm40.metric	Yes

Class Name (in Alphabetic Order)	Java Package	Can be Instantiated?
ArmMetricDefinition	org.opengroup.arm40.metric	No
ArmMetricFactory	org.opengroup.arm40.metric	Yes
ArmMetricGauge32	org.opengroup.arm40.metric	Yes
ArmMetricGauge32Definition	org.opengroup.arm40.metric	Yes
ArmMetricGauge64	org.opengroup.arm40.metric	Yes
ArmMetricGauge64Definition	org.opengroup.arm40.metric	Yes
ArmMetricGaugeFloat32	org.opengroup.arm40.metric	Yes
ArmMetricGaugeFloat32Definition	org.opengroup.arm40.metric	Yes
ArmMetricGroup	org.opengroup.arm40.metric	Yes
ArmMetricGroupDefinition	org.opengroup.arm40.metric	Yes
ArmMetricNumericId32	org.opengroup.arm40.metric	Yes
ArmMetricNumericId32Definition	org.opengroup.arm40.metric	Yes
ArmMetricNumericId64	org.opengroup.arm40.metric	Yes
ArmMetricNumericId64Definition	org.opengroup.arm40.metric	Yes
ArmMetricString32	org.opengroup.arm40.metric	Yes
ArmMetricString32Definition	org.opengroup.arm40.metric	Yes
ArmPrestartTimeStats	org.opengroup.arm40.transaction	Yes
ArmSystemAddress	org.opengroup.arm40.tranreport	Yes
ArmTimestamp	org.opengroup.arm40.transaction	No
ArmTimestampOpaque	org.opengroup.arm40.transaction	Yes
ArmTimestampStrings	org.opengroup.arm40.transaction	Yes
ArmTimestampUsecJan1970	org.opengroup.arm40.transaction	Yes
ArmToken	org.opengroup.arm40.transaction	No
ArmTranReport	org.opengroup.arm40.tranreport	Yes
ArmTranReportFactory	org.opengroup.arm40.tranreport	Yes
ArmTranReportWithMetrics	org.opengroup.arm40.tranreport	Yes
ArmTransaction	org.opengroup.arm40.transaction	Yes
ArmTransactionControl	org.opengroup.arm40.transaction	Yes
ArmTransactionDefinition	org.opengroup.arm40.transaction	Yes
ArmTransactionFactory	org.opengroup.arm40.transaction	Yes
ArmTransactionDefinitionControl	org.opengroup.arm40.transaction	Yes
ArmTransactionWithMetrics	org.opengroup.arm40.metric	Yes
ArmTransactionWithMetricsDefinition	org.opengroup.arm40.metric	Yes

Class Name (in Alphabetic Order)	Java Package	Can be Instantiated?
ArmUser	org.opengroup.arm40.transaction	Yes

12.3 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, ArmSystemAddress, ArmID), 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.

12.4 org.opengroup.arm40.transaction.ArmApplication

ArmApplication represents an instance of an executing application. It provides an anchor point for associating ArmTransaction objects with the application instance. Instances of ArmApplication are created using the **newArmApplication()** method of ArmTransactionFactory. It has the following attributes, all of which are immutable:

- Application definition. The metadata common to all instances of this application. The value must not be null.
- Group (optional). A group is a set of application instances that are treated as a group for some aspects of management, such as workload balancing (work could be routed to any instance of the group). Application instances for a given software product that are started for a common runtime purpose are typically very good candidates for using the same group name. For example, identical replica instances of a product started across multiple processes or servers to address a specific transaction workload objective can be, advantageously to the ARM agent, commonly identified by the group name. The maximum length is 255 characters. The value may be null. A non-null value should not contain trailing blank characters or consist of only blank characters.
- Instance (optional). The instance could be used to distinguish between instances of the same application. ARM does not require that this field be used or that it be unique, although its use is suggested. The maximum length is 255 characters. The value may be null. A non-null value should not contain trailing blank characters or consist of only blank characters.
- Context values (optional). The "value" part of (name,value) properties that may vary per application instance. The "name" part is available via **getDefinition().getIdentityProperties().getContextName()**. The values are position-sensitive – they match the position in the referenced context name array. The context property name at the specified array index must have been set to a non-null value when the ArmApplicationDefinition object was created. If the name is null or a zero-length string, both the name and value are ignored. If the value is null or a zero-length string, the meaning is that there is no value for this instance. Names should not contain trailing blank characters or consist of only blank characters.

  Note that both ArmApplication and ArmTransaction have context values that may be unique for each application and transaction, respectively. However, the mechanisms for setting the context values are different. In ArmApplication, they are set in the factory method and are immutable afterwards. In ArmTransaction, they are set at any time and processed each time a **start()** executes.

  The reason for the difference is that ArmApplication objects are not re-used. One object is created per running application. If another application starts running, a new ArmApplication is created. This is a reasonable design pattern because applications are long-lived. In most cases, once an application starts, it continues to execute for a long time, so the cost of creating the ArmApplication object and then garbage collecting it is not significant.

Transactions, on the other hand, are often very short-lived, with response times that are often measured in milliseconds. Creating a new ArmTransaction object each time a transaction executes, and then garbage collecting it afterwards, would result in unacceptable and unnecessary overhead. Instead, an ArmTransaction object is created and then re-used over and over. Each executing transaction is represented by the paired **start()**/**stop()** methods. For this reason, context values are set using setter methods.

end() indicates that the application has halted. After **end()** executes, the application must not call any other method of the ArmApplication object, it may not use any reference to the ArmApplication object, nor may it call any method of any object created using a reference to the ArmApplication object (e.g., creating an instance of ArmTransaction using the **newArmTransaction()** method of ArmTransactionFactory). Any transactions that are currently in-process [**start()** executed but **stop()** not executed] will be discarded by implicitly executing the ArmTransaction **reset()** method. The ARM implementation should protect itself against a poorly behaved application that does not respect the specification.

getApplicationControl() and **getTransactionDefinitionControl()** may be used by applications to understand the ARM implementation's preferences for instrumentation details that apply at the scope of all transactions in the entire application or all transactions of a registered transaction ID, respectively. The getter methods in the returned control blocks, if any, may be queried to learn the details. If null is returned, the application should use its default level of instrumentation. It is a good practice to use **getApplicationControl()** and **getTransactionDefinitionControl()** periodically, such as after every 1000 transactions or every 15 minutes, to refresh the settings.

```
public interface ArmApplication extends ArmInterface {
// No Public Constructors
// Public Instance Methods
    public int end();
    public String getContextValue(int index);
    public ArmApplicationDefinition getDefinition();
    public String getGroup();
    public ArmApplicationControl getApplicationControl();         //4.1
    public String getInstance();
    public ArmTransactionDefinitionControl
getTransactionDefinitionControl(ArmID tranID); //4.1
}
```

12.5 org.opengroup.arm40.transaction.ArmApplicationControl

[New in ARM 4.1]

ArmApplicationControl is used by applications to request the type and scope of instrumentation the ARM implementation prefers. Its use is optional for both applications and ARM implementations. Further, the control settings represent preferences; they are not binding on the application.

The scope of the settings applies to all transactions registered by this application. Some of these settings may be overridden by ArmTransactionDefinitionControl and/or ArmTransactionControl.

ArmApplicationControl is created using ArmApplication's **getApplicationControl()** method. Once created, the values are immutable. It is a good practice to periodically get a new ArmApplicationControl, and discard the old one, to see if there have been any changes.

The core instrumentation capabilities that should always be used if there is any instrumentation are the following:

- ArmTransaction's **start()**, **update()**, and **stop()**
- ArmTransaction's parent correlator and current correlator
- Application and transaction identity and context

getCollectionDepth() indicates the granularity of transactions that the ARM implementation prefers. This setting is orthogonal to other control settings, such as **isBindThreadRequested()**. It influences only whether and how often start/update/stop API calls are used. The four possible values have the further semantic knowledge:

- org.opengroup.arm40.transaction.ArmConstants.COLLECTION_DEPTH_NONE

  The implementation prefers that the application does not measure any transactions.

- org.opengroup.arm40.transaction.ArmConstants.COLLECTION_DEPTH_PROCESS

  The implementation prefers that the application measures these transactions at a process granularity; i.e., that it use a single **start()**/**stop()** pair per process.

- org.opengroup.arm40.transaction.ArmConstants.COLLECTION_DEPTH_CONTAINER

  The implementation prefers that the application measures these transactions at a "container" granularity; i.e., that it use a single **start()**/**stop()** pair per container.

  The determination of what constitutes a container is at the discretion of the application architect who designs the ARM instrumentation. For many applications, the only logical container boundary is also the process boundary, in which case there would be no difference in how the settings of 1 and 2 would be interpreted. For other applications – such as complex middleware that may contain a web server, J2EE container, JDBC connectors – there could be multiple containers running within the same process.

- org.opengroup.arm40.transaction.ArmConstants.COLLECTION_DEPTH_MAX

  The implementation prefers that the application measures transactions at the maximum possible granularity. This may be the same as the process or container granularity, depending on the application.

isTransactionDefinitionControlUsed() indicates whether the ARM implementation uses ArmTransactionDefinitionControl. If False, ArmApplication's **getTransactionDefinitionControl()** method will return null, regardless of the input transaction ID.

isTransactionControlUsed() indicates whether the ARM implementation uses ArmTransactionControl. If False, ArmTransaction's **getControl()** method will return null in all cases.

isPrivateDataRequested() indicates whether private data (e.g., account numbers) should be provided in any form, such as in a metric or diagnostic data, or in a correlator. The application determines what constitutes private data.

isSecureDataRequested() indicates whether secure data (e.g., passwords) should be provided in any form, such as in a metric or diagnostic data, or in a correlator. The application determines what constitutes secure data.

isBindThreadRequested() indicates that the implementation prefers that the application uses ArmTransaction's **bindThread()** and **unbindThread()** calls.

isBlockRequested() indicates that the implementation prefers that the application uses ArmTransaction's **blocked()** and **unblocked()** calls.

isDiagnosticDataRequested() indicates that the implementation prefers that the application use ArmDiagnosticProperties or the diagnostic detail string on ArmTransaction's **stop()**.

isMessageEventDataRequested() indicates that the implementation prefers that the application informs it about message exchanges and other asynchronous flows.

isMetricDataRequested() indicates that the implementation prefers that the application provides ArmMetric data.

isUserDataRequested() indicates that the implementation prefers that the application uses ArmTransaction's **setUser()**.

```
public interface ArmApplicationControl extends ArmInterface {
// New in ARM 4.1
// No Public Constructors
// Public Instance Methods
    public int     getCollectionDepth();
    public boolean isTransactionDefinitionControlUsed();
    public boolean isTransactionControlUsed();
    public boolean isPrivateDataRequested();
    public boolean isSecureDataRequested();
    public boolean isBindThreadRequested();
    public boolean isBlockRequested();
    public boolean isDiagnosticDataRequested();
    public boolean isMessageEventDataRequested();
```

```
    public boolean isMetricDataRequested();
    public boolean isUserDataRequested();
}
```

12.6 org.opengroup.arm40.transaction.ArmApplicationDefinition

ArmApplicationDefinition describes the attributes of an application that do not change from one instance of the application to another. It provides an anchor point for associating ArmTransactionDefinition and ArmMetricDefinition objects with the application. It is created with the **newArmApplicationDefinition()** method of ArmTransactionFactory. It has the following attributes, all of which are immutable:

- Name. The maximum length is 127 characters (CIM allows 256 but ARM 4.0 C Bindings allow 128 characters, including the null-termination character, so 127 is used). The name must not be null or zero-length. A name should be chosen that is unique, so generic names that might be used by a different development team, such as "Payroll Application", should not be used. Names should not contain trailing blank characters or consist of only blank characters.

- (optional) Identity property names and values and context property names in arrays. See the discussion of identity and context property names in ArmIdentityProperties.

- (optional) ID. An optional 16-byte ID may be associated with the identity of an application definition. The returned value, which could be null, is the same value passed to the **newArmApplicationDefinition()** method of ArmTransactionFactory. The ID value is bound to a unique combination of the application name, any identity property names and values, and any context property names. When provided, the ID may be used as a concise alias for the unique combination. It may be null.

destroy() does not, of course, destroy the ArmApplicationDefinition object. It does signal to the ARM implementation that the definition and all related definitions (e.g., ArmTransactionDefinition) within its scope are no longer needed. The normal behavior would be for the ARM implementation to release its references to all those objects. If the application also releases its references, the objects would be eligible for garbage collection. After **destroy()** is called, no method on any object that is scoped by the ArmApplicationDefinition should be called again. If a method is called, the results are unpredictable.

```
public interface ArmApplicationDefinition extends ArmInterface {
// No Public Constructors
// Public Instance Methods
    public void destroy();
    public String getName();
    public ArmIdentityProperties getIdentityProperties();
    public ArmID getID();
}
```

12.7 org.opengroup.arm40.tranreport.ArmApplicationRemote

ArmApplicationRemote represents an instance of an application executing on a remote system. It differs from ArmApplication in that ArmApplication represents an application executing on the local system.

ArmApplicationRemote provides an anchor point for associating ArmTranReport objects with a system's network address. Instances of ArmApplicationRemote are created using the **newArmApplicationRemote()** method of ArmTranReportFactory. It adds the following attribute to those in ArmApplication:

- System address. The system address is the address (format and address byte array) of the system.

```
public interface ArmApplicationRemote extends ArmApplication {
// No Public Constructors
// Public Instance Methods
    public ArmSystemAddress getSystemAddress();
}
```

12.8 org.opengroup.arm40.transaction.ArmBlockCause

ArmBlockCause describes the cause of a blocking condition. Its use is optional. When used, it is passed on ArmTransaction's **blocked()**.

The Cause ID *must* be one of the following values. All other values are reserved.

- org.opengroup.arm40.transaction.ArmConstants.BLOCK_CAUSE_SYNCHRONOUS_EVENT
- org.opengroup.arm40.transaction.ArmConstants.BLOCK_CAUSE_ASYNCHRONOUS_EVENT

No Extended Cause ID values are currently defined. The range of possible values ais re partitioned as follows:

- All non-negative values are reserved by the standard.
- All negative values are not defined by the standard. There are no restrictions on who uses these values or what they mean.

The Description is a String with a maximum of 127 characters.

```
public interface ArmBlockCause extends ArmInterface {
// New in ARM 4.1
// No Public Constructors
// Public Instance Methods
    public String getDescription();
    public int    getExtendedCause();
    public int    getCause();
    public int    setDescription(String desc);
    public int    setExtendedCause(int extendedCauseID);
    public int    setCause(int causeID);
}
```

12.9 org.opengroup.arm40.transaction.ArmConstants

ArmConstants are constants that are widely used in the ARM 4.0 interfaces. They are declared in one interface as a matter of convenience (instead of duplicating them in multiple interfaces).

```
public interface ArmConstants {
// No Public Constructors
// Constants

    // New in ARM 4.1
    public static final int BLOCK_CAUSE_SYNCHRONOUS_EVENT;
        // A synchronous event caused the block        (=1)
    public static final int BLOCK_CAUSE_ASYNCHRONOUS_EVENT;
        // An asynchronous event caused the block      (=2)

    // New in ARM 4.1
    public static final int COLLECTION_DEPTH_NONE;
        // Collection depth = None                     (=0)
    public static final int COLLECTION_DEPTH_PROCESS;
        // Collection depth = Process granularity      (=1)
    public static final int COLLECTION_DEPTH_CONTAINER;
        // Collection depth = Container granularity    (=2)
    public static final int COLLECTION_DEPTH_MAX;
        // Collection depth = Maximum granularity      (=3)

    public static final int CORR_MAX_LENGTH;
        // Max length of a correlator (currently = 512 bytes)
    public static final int CORR_MIN_LENGTH;
        // Minimum length of a correlator (= 4 bytes)

    public static final int DIAG_DETAIL_MAX_LENGTH;
        // Maximum length of diagnostic detail string
        // (currently = 4095 characters)

    public static final int ID_LENGTH;
        // Length of all IDs (16 bytes)

    // New in ARM 4.1
    public static final int MESSAGE_EVENT_MAX_COUNT;
        // Max number of events in ArmMessageEventGroup (currently=32)

    public static final int METRIC_MAX_COUNT;
        // Max number of metric slots (currently = 7)
    public static final int METRIC_MAX_INDEX;
        // Max index of a metric slot (currently = 6)
    public static final int METRIC_MIN_INDEX;
        // Min index of a metric slot (currently = 0)

    public static final int NAME_MAX_LENGTH;
        // Max chars in app/tran/metric name (currently = 127)

    public static final int PROPERTY_MAX_COUNT;
```

```
        // Maximum number of identity and context properties
        // (currently = 20)
    public static final int PROPERTY_MAX_INDEX;
        // Maximum array index of an identity or context property
        // (currently = 19)
    public static final int PROPERTY_MIN_INDEX;
        // Minimum array index of an identity or context property
        // (currently = 0)
    public static final int PROPERTY_NAME_MAX_LENGTH;
        // Max chars in an identity or context property
        // (currently = 127)
    public static final int PROPERTY_URI_MAX_LENGTH;
        // Maximum chars in an URI property (currently = 4095)
    public static final int PROPERTY_VALUE_MAX_LENGTH;
        // Max chars in an identity or context property
        // (currently = 255)

    public static final int STATUS_ABORT;
        // Valid status value for ArmTranReport and ArmTransaction (=1)
    public static final int STATUS_FAILED;
        // Valid status value for ArmTranReport and ArmTransaction (=2)
    public static final int STATUS_GOOD;
        // Valid status value for ArmTranReport and ArmTransaction (=0)
    public static final int STATUS_INVALID;
        // Status value used when appl. passes an invalid value (=-1)
        // or if status is requested before it has ever been set.
    public static final int STATUS_UNKNOWN;
        // Valid status value for ArmTranReport and ArmTransaction (=3)

    public static final int USE_CURRENT_TIME;
        // Used with ArmTranReport (= -1)
// No Instance Methods
}
```

12.10 org.opengroup.arm40.transaction.ArmCorrelator

ArmCorrelator represents a correlation token passed from a calling transaction to a called transaction. The correlation token may be used to establish a calling hierarchy across processes and systems. 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 each executing transaction. Applications do not need to understand correlator internals. See Appendix B for more information about correlator formats.

A correlation token is a maximum of 512 bytes, including the header.

ArmCorrelator is created in one of three ways:

- The **newArmCorrelator()** method of ArmTransactionFactory takes as input a byte array in network byte order, such as would be received from a caller.
- The **getCorrelator()** method of ArmTransaction creates a correlator object for the currently or most recently executed transaction.
- The **generateCorrelator()** method of ArmTranReport creates a correlator object for what is presumed to be the next transaction to be executed.

An application may extract the byte array in network byte order, which is the format needed to send to a called transaction, using the **copyBytes()** or **getBytes()** methods of ArmToken, ArmCorrelator's parent interface.

ARM 4.1 introduced the "asynchronous" and "independent transaction" flags. These are notably different from the "agent trace" and "application traceT flag because the values are set directly by the instrumented application. A typical sequence would be the following:

```
ArmTransaction tran; ArmCorrelator corr;
corr   = tran.getCorrelator();
status = corr.setAsynchronous();
status = corr.setIndependentTran();
// pass correlator to child transactions…
```

Interface methods:

- **isAgentTrace()** indicates whether the "agent trace" flag is on in the correlator.
- **isApplicationTrace()** indicates whether the "application trace" flag is on in the correlator.
- [ARM 4.1] **isAsynchronous()** indicates whether the "asynchronous" flag is on in the correlator.
- [ARM 4.1] **isIndependentTran()** indicates whether the "independent transaction" flag is on in the correlator.
- [ARM 4.1] **setAsynchronous()** causes the "asynchronous" flag to be set to the specified Boolean value.

- [ARM 4.1] **setIndependentTran()** causes the "independent transaction" flag to be set to the specified Boolean value.

```
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 boolean isAgentTrace();
    public boolean isApplicationTrace();
    public boolean isAsychronous();                              //4.1
    public boolean isIndependentTran();                          //4.1
    public int     setAsychronous(boolean b);                    //4.1
    public int     setIndependentTran(boolean b);                //4.1
}
```

12.11 org.opengroup.arm40.transaction.ArmDiagnosticProperties

[New in ARM 4.1.]

ArmDiagnosticProperties is new in ARM 4.1. It is used with ArmTransaction's **stop()** and ArmTranReport's **report()** method to pass a set of (name,value) pairs when a transaction stops. It differs from ArmIdentityProperties primarily in that both the name and value parameters are mutable, and there is no concept of the properties being arrays. The interface is identical to ArmProperties.

ArmDiagnosticProperties represents a set of constraints of the number and size of the property names and values.

Constraint Description	Constraint Value
Maximum number of non-null properties	20
Maximum number of characters of any property name	127
Maximum number of characters of any property name + property value	2046
Maximum number of characters of all property names + property values	4056

```
public interface ArmDiagnosticProperties extends ArmProperties {
// New in ARM 4.1
// No Public Constructors
// No new public instance methods are added. ArmDiagnosticProperties
//   has the same interface as ArmProperties, but it represents a set
//   of constraints on the property names and values.
}
```

12.12 org.opengroup.arm40.transaction.ArmErrorCallback

The use of ArmErrorCallback is *optional*.

ArmErrorCallback is different from all the other ARM interfaces because instead of the ARM implementation creating objects that implement the interface, the *application* creates an object that implements it. The application can create an ArmErrorCallback and register it with the **setErrorCallback()** method of ArmTransactionFactory, ArmTranReportFactory, or ArmMetricFactory. If the registration is accepted, anytime a method results in an error, the ARM implementation:

- Sets the object's error code; this error code can be retrieved with the **getErrorCode()** method of ArmInterface, from which all other ARM interfaces with methods derive
- Calls **errorCodeSet()** in ArmErrorCallback, if an ArmErrorCallback has been successfully registered
- Returns the error code if the method that caused the error returns an error code

There are no expected or required behaviors for the implementation of **errorCodeSet()**. Examples of things a callback method may do are:

- Nothing
- Log an error; this would be a common behavior during program debugging
- Reset the error code to zero, or set it to some other value; this is a way for an error handling policy to be implemented centrally, and communicated to the rest of the application code

When **errorCodeSet()** is called, all the parameters must contain non-null values.

- errorObject is a reference to the ARM implementation's object that detected the error. The callback method can use the errorObject **getErrorCode()** to get the error code value, which will be negative.
- interfaceName is the name of an interface in one of the ARM specification packages.
- methodName is the name of a method in that interface.

```
public interface ArmErrorCallback {
// No Public Constructors
// Public Instance Methods
    public void errorCodeSet(ArmInterface errorObject,
        String interfaceName, String methodName);
}
```

12.13 org.opengroup.arm40.transaction.ArmID

ArmID implements an immutable wrapper around a 16-byte ID. IDs may be used to identify metadata about applications, transactions, metrics, systems, and users. The ID may be a standard DCE UUID (universally unique identifier) but need not be. Any unique 16-byte value will suffice. There is no central registry of IDs that would guarantee uniqueness. Programs creating these IDs are expected to use an algorithm that will take advantage of the available 128 bits to create an ID for which there will be a vanishingly small probability of it being a duplicate of an ID created by another program.

It is created with the **newArmID()** method of ArmTransactionFactory, whose input is the 16 bytes in a byte array.

This interface was named ArmUUID in ARM 3.0. There are no other changes for ARM 4.0.

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

12.14 org.opengroup.arm40.transaction.ArmIdentityProperties

ArmIdentityProperties is new in ARM 4.0. It addresses a requirement to accept a set of string (name,value) pairs that extend the concept of application and transaction identity and context.

ARM defines two types of properties – identity and context. The difference between them is as follows:

- An identity property's name and value are the same for all instances of an application or transaction.
- A context property's name is the same for all instances of an application or transaction, but a context property's value may vary for each instance.

ArmIdentityProperties contains some attributes that are common to the identity of both an application and a transaction. The full identity is captured in ArmApplicationDefinition and ArmTransactionDefinition, which each adds other attributes to those listed here. All these attributes are immutable. The common attributes include:

- A set of identity names (maximum of 20; maximum length of 127 characters)
- A set of identity values (maximum of 20; maximum length of 255 characters)
- A set of context names (maximum of 20; maximum length of 127 characters). The context values may change with each instance so they are not part of the identity. The context values are in ArmApplication or ArmTransaction.

Property names and values are in arrays. A property name and value form a (name,value) pair. The array index of a property value in the value array is bound to the property name at the same index in the name array. Moving the (name,value) pair to a different index does not affect the identity of the application. For example, if an application is registered once with a name A and a value X in array indices 0 and once with the same name and value in array indices 1, the registered identity has not changed.

If a name is repeated in the array, the name and its corresponding value are ignored, and the first instance of the name in the array (and its corresponding value) is used. If the pointer is null or points to a zero-length string, the (name,value) pair is ignored. Names should not contain trailing blank characters or consist of only blank characters.

The names of identity and context properties can be any string, with one exception. Strings beginning with the four characters "ARM:" are reserved for the ARM specification. The specification will define names with known semantics using this prefix. One name format is currently defined. Any name beginning with the eight-character prefix "ARM:CIM:" represents a name defined using the DMTF CIM (Distributed Management Task Force Common Information Model) naming rules. For example, "ARM:CIM:CIM_SoftwareElement.Name" indicates that the property value has the semantics of the Name property of the CIM_SoftwareElement class. It is anticipated that additional naming semantics are likely to be added in the future.

ArmIdentityProperties is created with the **newArmIdentityProperties()** method of ArmTransactionFactory.

Interface methods:

- **getIdentityName()** returns the string (the name part of the (name,value) identity property) at the specified array index. The returned value will be null if either the name or value at the index is set to null.
- **getIdentityValue()** returns the string (the value part of the (name,value) identity property) at the specified array index. The returned value will be null if either the name or value at the index is set to null.
- **getContextName()** returns the string (the name part of the (name,value) context property) at the specified array index. The returned value may be null.

```
public interface ArmIdentityProperties extends ArmInterface {
// No Public Constructors
// Public Instance Methods
    public String getIdentityName(int index);
    public String getIdentityValue(int index);
    public String getContextName(int index);
}
```

12.15 org.opengroup.arm40.transaction.ArmIdentityPropertiesTransaction

ArmIdentityPropertiesTransaction extends ArmIdentityProperties for transactions by adding a URI property. Unlike the other identity properties, the URI property is implicitly named.

Like other identity properties, it is the same for all instances of the same transaction, and is immutable. In practice, this means that the URI value may be truncated from the URI that is actually used. In particular, parameters that are appended to the end of a URI will often be truncated because they are often different each time the URI is invoked. In this case, the parameters might be provided as part of the URI context value of an ArmTransaction object.

ArmIdentityPropertiesTransaction is created using the **newArmIdentityPropertiesTransaction()** method of ArmTransactionFactory.

Interface methods:

- **getURIValue()** returns the string representing the URI value, if any. The maximum length is 4095 characters. The returned value will be null if the value is set to null.

```
public interface ArmIdentityPropertiesTransaction extends
    ArmIdentityProperties {
// No Public Constructors
// Public Instance Methods
    public String getURIValue();
}
```

12.16 org.opengroup.arm40.transaction.ArmInterface

ArmInterface is the root of the inheritance hierarchy for almost all ARM interfaces. It provides a common way to handle errors. If a method invocation on any ARM object causes an error, the error code returned by the object's **getErrorCode()** will be negative. If no error occurs, the error code is zero. Several methods also return the error code as an int return value. If an error occurs in a factory method (e.g., a method in ArmTransactionFactory), the error code is set in both the factory object and the newly created object. The implementation of the object has sole discretion as to whether a method results in an error.

The error code may change any time a method of the object is executed. Executing a method overrides the previous error code value. The only methods that will never change the error code are **getErrorCode()** and **getErrorMessage(**). If multiple threads are processing the same object simultaneously, the results are unpredictable. An error code set as a result of an operation in method X could be replaced by an operation in method Y before thread X executes **getErrorCode()**.

The error code can also be set with **setErrorCode()**. **setErrorCode()** is a way for the application to reset or change the error code. It would most typically be used in a callback registered with the **setErrorCallback()** method of ArmTransactionFactory, ArmTranReportFactory, or ArmMetricFactory. In this way, the application can implement a central error handling policy in the callback. A value set with **setErrorCode()** in a callback overwrites the previous value (i.e., the value that caused the callback to be made).

For any non-zero error code returned by an object, the application can request from the same object a string message describing the error using **getErrorMessage()**. If the object does not support the function or does not recognize the error code, it returns null.

```
public interface ArmInterface {
// No Public Constructors
// Public Instance Methods
    public int getErrorCode();
    public void setErrorCode(int errorCode);
    public String getErrorMessage(int errorCode);
}
```

12.17 org.opengroup.arm40.transaction.ArmMessageEvent

ArmMessageEvent is a superclass for all the message event type interfaces. The common behaviors of all message events are:

- Each event has a description that can be read and written using **getDescription()** and **setDescription()**.
- Each event can indicate if it is a Received Event or Sent Event using **isMessageReceivedEvent()** or **isMessageSentEvent()**.

```
public interface ArmMessageEvent extends ArmInterface {
// New in ARM 4.1
// No Public Constructors
// Public Instance Methods
    public String  getDescription();
    public boolean isMessageReceivedEvent();
    public boolean isMessageSentEvent();
    public int     setDescription(String desc);
}
```

12.18 org.opengroup.arm40.transaction.ArmMessageEventGroup

ArmMessageEventGroup is a container for an array of ArmMessageEvent instances, plus an End of Flow indicator. It provides a convenient aggregation for providing the set of events to ArmTransaction.

The events are represented as an array of a maximum of 64 ArmMessageEvent instances. There is a further constraint that there can be no more than 32 send events and 32 receive events. For example, there could be 32 send events plus 10 receive events for a total of 42 events, but not 33 send events plus 10 receive events, even though the total number of events (43) is less than 64.

setEvent() inserts an event in the array at the specified index. **setEvent(**index, null**)** clears the event at that index.

getEvent() retrieves an event at the index.

clearAllEvents() is equivalent to executing **setEvent(**index, null**)** for all possible index positions.

setEndOfFlow(True) indicates that the currently executing transaction represents the last processing step in a compound transaction that started in another process. In this case, the stop time of the currently executing transaction can be considered the stop time for the entire compound transaction. The default value is False for each transaction unless it is set to True with this method. It serves no purpose to set this flag to True and pass it to ArmTransaction more than once for a given transaction. **setEndOfFlow(**False**)** indicates that this condition does not exist, or at least is not known to exist.

isEndOfFlow() returns the current value.

```
public interface ArmMessageEventGroup extends ArmInterface {
// New in ARM 4.1
// No Public Constructors
// Public Instance Methods
    public int             clearAllEvents();
    public ArmMessageEvent getEvent(int index);
    public boolean         isEndOfFlow();
    public int             setEndOfFlow(boolean b)
    public int             setEvent(int index, ArmMessageEvent e);
}
```

12.19 org.opengroup.arm40.transaction.ArmMessageReceivedEvent

ArmMessageReceivedEvent describes a received message. Each message has an optional description (described in ArmMessageEvent) and an optional received ArmCorrelator, if one was received with the message. The correlator is useful for indicating the transaction and message originator to which this message event relates. A null pointer indicates that no correlator is provided.

Use **setRecievedCorrelator()** to set the correlator's value, and **getReceivedCorrelator()** to retrieve the previously set value. **setRecievedCorrelator(**null**)** clears the correlator value to null.

A restriction is that there can be no more than 32 received correlators associated with a transaction, including the parentCorr parameter of ArmTransction's **start()** plus any correlators passed in ArmMessageReceivedEvent. For example, if a correlator is passed as a parameter on **start()**, only 31 other correlators may be passed in an ArmMessageReceivedEvent for this transaction. Any additional correlators are ignored. There are no constraints on the uniqueness of the received correlators or any requirements that either the application or the ARM implementation test the contents of each correlator to see if it is unique.

Applications that use ARM 4.1 now have two ways to provide parent correlators: the parentCorr parameter of ArmTransction's **start()** and ArmMessageReceivedEvent. The application designer should consider the following when deciding which to use. The contents of a parent correlator often influence the contents of a transaction's current correlator. For example, information about the originating user transaction (the root parent of the call graph) may be copied from the parent correlator to each child correlator at each node in the call graph, so they are available at all the nodes. Because a current correlator is created when **start()** executes, only parent correlators available at that time influence the contents of the current correlator.

The recommended practice for correlators available when **start()** executes is the following. Correlators received after **start()** executes can only be passed in ArmMessageReceivedEvent.

- If there is one parent correlator available when **start()** executes, pass the correlator in the parentCorr parameter.
- If there are multiple parent correlators available when **start()** executes, and one of them is considered the primary parent, pass that one in the parentCorr parameter and the other(s) in ArmMessageReceivedEvent.
- If there are multiple parent correlators available when **start()** executes and none are considered the most important, pass all the correlators in ArmMessageReceivedEvent.

```
public interface ArmMessageReceivedEvent extends ArmMessageEvent {
// New in ARM 4.1
// No Public Constructors
// Public Instance Methods
    public ArmCorrelator getCorrelatorReceived();
    public int           setCorrelatorReceived(ArmCorrelator corr);
}
```

12.20 org.opengroup.arm40.transaction.ArmMessageSentEvent

ArmMessageSentEvent describes one or more sent messages with an optional description (described in ArmMessageEvent) and an optional count of messages sent that are considered equivalent to each other. The equivalency is at the discretion of the instrumenter. A value of 0 is treated as the default value of 1. The counter provides a way for an application to indicate multiple messages of the same type with a single ArmMessageSentEvent.

Use **setMessageSentCount()** to set the count, and **getMessageSentCount()** to retrieve the previously set value.

```
public interface ArmMessageSentEvent extends ArmMessageEvent {
// New in ARM 4.1
// No Public Constructors
// Public Instance Methods
    public int getMessageSentCount();
    public int setMessageSentCount(int count);
}
```

12.21 org.opengroup.arm40.metric.ArmMetric

ArmMetric is a superclass for all the metric interfaces. The common behavior of all metric subclasses is:

- **getDefinition()** returns the descriptive metadata (name, units, usage, ID) about the metric. The returned object will be the appropriate subclass of ArmMetricDefinition.
- Each subclass is also expected to implement **get()** and **set()** methods that take and/or return data of the appropriate type for the subclass. These methods are not defined in this interface because they have different signatures, depending on the type of metric.

Objects that implement a subclass of ArmMetric are used with ArmTransactionWithMetrics and ArmTranReportWithMetrics. They are bound via ArmMetricGroup when the ArmTransactionWithMetrics or ArmTranReportWithMetrics instance is created. Each ArmMetric instance can be bound to any number of transactions. Setting the value of the ArmMetric instance effectively sets the value for all the transactions to which it is bound. The value affects each ArmTransactionWithMetrics instance the next time a **start()**, **update()**, or **stop()** is executed on the instance. The value affects each ArmTranReportWithMetrics instance the next time a **report()** is executed on the instance.

```
public interface ArmMetric extends ArmInterface {
// No Public Constructors
// Public Instance Methods
    public ArmMetricDefinition getDefinition();
}
```

12.22 org.opengroup.arm40.metric.ArmMetricCounter32

ArmMetricCounter32 implements a 32-bit integer counter. It is the same as ARM 2.0 metric type=1 (ARM_Counter32).

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

12.23 org.opengroup.arm40.metric.ArmMetricCounter32Definition

ArmMetricCounter32Definition is a subclass of ArmMetricDefinition and serves as a marker interface that binds the metadata in ArmMetricDefinition to the metric data type, and describes an object interface that can be instantiated (there are no factory methods for objects that implement only ArmMetricDefinition). No new methods beyond those in ArmMetricDefinition are added.

```
public interface ArmMetricCounter32Definition extends
    ArmMetricDefinition {
// No Public Constructors
// No new Public Instance Methods.
// All methods are inherited from ArmMetricDefinition.
}
```

12.24 org.opengroup.arm40.metric.ArmMetricCounter64

ArmMetricCounter64 implements a 64-bit integer counter. It is the same as ARM 2.0 metric type=2 (ARM_Counter64).

```
public interface ArmMetricCounter64 extends ArmMetric {
// No Public Constructors
// Public Instance Methods
    public long get();
    public int set(long value);
}
```

12.25 org.opengroup.arm40.metric.ArmMetricCounter64Definition

ArmMetricCounter64Definition is a subclass of ArmMetricDefinition and serves as a marker interface that binds the metadata in ArmMetricDefinition to the metric data type, and describes an object interface that can be instantiated (there are no factory methods for objects that implement only ArmMetricDefinition). No new methods beyond those in ArmMetricDefinition are added.

```
public interface ArmMetricCounter64Definition extends
    ArmMetricDefinition {
// No Public Constructors
// No new Public Instance Methods.
// All methods are inherited from ArmMetricDefinition.
}
```

12.26 org.opengroup.arm40.metric.ArmMetricCounterFloat32

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 the C bindings for ARM 2.0 and ARM 4.0, a floating-point value is provided directly. This was not done with the C bindings because ARM would have to support multiple floating-point formats, depending on the programming language and/or machine architecture, and the complexity was not deemed worthwhile.

```
public interface ArmMetricCounterFloat32 extends ArmMetric {
// No Public Constructors
// Public Instance Methods
    public float get();
    public int set(float value);
}
```

12.27 org.opengroup.arm40.metric.ArmMetricCounterFloat32Definition

ArmMetricCounterFloat32Definition is a subclass of ArmMetricDefinition and serves as a marker interface that binds the metadata in ArmMetricDefinition to the metric data type, and describes an object interface that can be instantiated (there are no factory methods for objects that implement only ArmMetricDefinition). No new methods beyond those in ArmMetricDefinition are added.

```
public interface ArmMetricCounterFloat32Definition extends
    ArmMetricDefinition {
// No Public Constructors
// No new Public Instance Methods.
// All methods are inherited from ArmMetricDefinition.
}
```

12.28 org.opengroup.arm40.metric.ArmMetricDefinition

ArmMetricDefinition is a superclass for all the metric definition interfaces. All the methods are defined in ArmMetricDefinition. The subclasses serve as markers for the data types.

All the publicly accessible attributes are immutable and have a getter method for them. The attributes are:

- The definition of the application that contains the transaction. It must not be null.
- Name. The maximum length is 127 characters. The name must not be null or zero-length.
- Units. An optional string describing the units of measurement, such as "files" or "jobs in queue". It may be null.
- Usage. Describes any additional information about the semantics of the metric. Most metrics will be classified as "general", indicating that there is no specific semantic declared. The two specific semantics are the size of the transaction – such as the number of files that were backed up – and a status that could contain data, such as an error code that describes why a transaction failed. See the ArmMetricDefinition interface for constants for the defined value. Any negative value is also permitted; the negative range is reserved for application-specific values.
- Optional ID. An optional 16-byte ID may be associated with the identity of a metric definition. The returned value, which could be null, is the same value passed to the factory method [e.g., the **newArmMetricCounter32Definition()** method of ArmMetricFactory]. The ID value is bound to a unique combination of the metric format (e.g., Counter32), name, usage, and unit properties. When provided, the ID may be used as a concise alias for the unique combination. It may be null.

```
public interface ArmMetricDefinition extends ArmInterface {
// No Public Constructors
// Constants
    public static final int METRIC_USE_GENERAL;
        // No usage semantics are declared.
    public static final int METRIC_USE_TRAN_SIZE;
        // Metric represents the "size" of the transaction
        // (counter and gauge only).
    public static final int METRIC_USE_TRAN_STATUS;
        // Metric represents status, like an error code
        // (numeric ID and string only).
// Public Instance Methods
    public ArmApplicationDefinition getApplicationDefinition();
    public ArmID getID();
    public String getName();
    public String getUnits();
    public short getUsage();
}
```

12.29 org.opengroup.arm40.metric.ArmMetricFactory

ArmMetricFactory provides methods to create instances of the classes in the org.opengroup.arm40.metric package.

Factory Version: The org.opengroup.arm40.metric package contains the interfaces defined in both ARM 4.0 and ARM 4.1. Applications that use ARM 4.1 could instantiate classes that implement ARM 4.0 but not ARM 4.1. See Section 8.2.1 for an explanation of how to handle this situation to avoid runtime errors.

newArmMetricCounter32Definition
newArmMetricCounter64Definition
newArmMetricCounterFloat32Definition
newArmMetricGauge32Definition
newArmMetricGauge64Definition
newArmMetricGaugeFloat32Definition
newArmMetricNumericId32Definition
newArmMetricString32Definition

Creates the metadata for the respective data type. All metrics have the same metadata. See the ArmMetricDefinition interface description for details.

newArmMetricGroupDefinition()

Creates an ordered set of ArmMetricDefinition subclasses ready for binding to transaction definition objects. The input is an array of ArmMetricDefinition objects. The ordering in the array is important. The array can have up to seven elements and is position-sensitive. To remain consistent with ARM 2.0, any ArmMetricDefinition subclass except ArmMetricString32Definition can be assigned to elements 0:5 and only ArmMetricString32Definition can be assigned to element 6.

Any element can be null. If the input array has fewer than seven elements, the rest of the elements are assigned a value of null. The array can be sparsely populated. For example, there can be a non-null ArmMetricDefinition reference in element 0 and 6, and null references in elements 1:5.

newArmTransactionWithMetricsDefinition()

Creates an object that contains the metadata about a transaction. Details about the metadata are found in the ArmTransactionDefinition section. In addition, a binding to a set of metadata about metrics is added via an ArmMetricGroupDefinition object.

newArmMetricCounter32
newArmMetricCounter64
newArmMetricCounterFloat32
newArmMetricGauge32
newArmMetricGauge64
newArmMetricGaugeFloat32
newArmMetricNumericId32
newArmMetricNumericId64
newArmMetricString32

Creates a metric of the specified type, using the input metadata.

newArmMetricGroup()

Creates an ordered set of ArmMetric subclasses ready for binding to transaction objects. The input is an array of ArmMetric objects plus a metric group definition object. The ordering in the array is important. The array can have up to seven elements and is position-sensitive. To remain consistent with ARM 2.0, any ArmMetric subclass except ArmMetricString32 can be assigned to elements 0:5 and only ArmMetricString32 can be assigned to element 6. The ArmMetricDefinition objects associated with each ArmMetric object must have the exact same values as the ArmMetricDefinition objects associated with the ArmMetricGroupDefinition object (they will often be the same objects, though this is not mandatory).

Any element can be null. If the input array has fewer than seven elements, the rest of the elements are assigned a value of null. The array can be sparsely populated. For example, there can be a non-null ArmMetric reference in element 0 and 6, and null references in elements 1:5.

newArmTranReportWithMetrics()

Creates an object that represents an instance of a transaction. The metadata about the transaction is supplied in an ArmTransactionWithMetricsDefinition object. An application instance (ArmApplication) provides a scoping context. The metrics are bound via an ArmMetricGroup object. If the metric group reference is null, the resulting ArmTranReportWithMetrics has no metrics, so it would have no more functions than an ArmTranReport object, except to return null to the **getMetricGroup()** method.

newArmTransactionWithMetrics()

Creates an object that represents an instance of a transaction. The metadata about the transaction is supplied in an ArmTransactionWithMetricsDefinition object. An application instance (ArmApplication) provides a scoping context. The metrics are bound via an ArmMetricGroup object. If the metric group reference is null, the resulting ArmTransactionWithMetrics has no metrics, so it would have no more functions than an ArmTransaction object, except to return null to the **getMetricGroup()** method.

Error Handling Philosophy

If an invalid set of parameters is passed to any method, such as an offset that extends beyond the end of an array, an object is returned that may contain dummy data. For example, a null byte[] addressBytes parameter might result in creating an object with an address of all 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, even if the data may be at least partially meaningless. The **getErrorCode()** method of each object can be used after it is created to test whether errors occurred. Refer to Chapter 9 for a more complete explanation.

ArmMetricFactory also serves as the anchor point for an application-registered callback function. **setErrorCallback()** is used to register a callback that will be called if any method of an object created by this factory object sets the error code to a negative value. The error code is retrieved using **getErrorCode()**, defined in ArmInterface, the root for most interfaces in the ARM specification. The boolean returned by **setErrorCallback()** indicates whether the registration is accepted. If an ARM implementation does not support the callback function, it will return false. **setErrorCallback(**null**)** unregisters any previously registered callback. Note that due to timing conditions or specifics of the ARM implementation, a previously registered callback may continue to be called for an indeterminate length of time after **setErrorCallback(**null**)** is executed. For a broader discussion of error handling, refer to Chapter 9.

Note that in the interface description below, the methods are not strictly alphabetical like they are with all other interfaces. This is because there are some logical groupings of methods, and it was felt that it would be more intuitive to present them in the groups.

```
public interface ArmMetricFactory extends ArmInterface {
// Public Constants
    public static final String propertyKey;
// Public Instance Methods
    /* --- metric definitions --- */
    public ArmMetricCounter32Definition
        newArmMetricCounter32Definition(
        ArmApplicationDefinition app,
        String name,
        String units,
        short usage,
        ArmID id);
    public ArmMetricCounter64Definition
        newArmMetricCounter64Definition(
        ArmApplicationDefinition app,
        String name,
        String units,
        short usage,
        ArmID id);
    public ArmMetricCounterFloat32Definition
        newArmMetricCounterFloat32Definition(
        ArmApplicationDefinition app,
        String name,
        String units,
        short usage,
        ArmID id);
    public ArmMetricGauge32Definition newArmMetricGauge32Definition(
        ArmApplicationDefinition app,
        String name,
        String units,
        short usage,
        ArmID id);
    public ArmMetricGauge64Definition newArmMetricGauge64Definition(
```

```
        ArmApplicationDefinition app,
        String name,
        String units,
        short usage,
        ArmID id);
    public ArmMetricGaugeFloat32Definition
        newArmMetricGaugeFloat32Definition(
        ArmApplicationDefinition app,
        String name,
        String units,
        short usage,
        ArmID id);
    public ArmMetricNumericId32Definition
        newArmMetricNumericId32Definition(
        ArmApplicationDefinition app,
        String name,
        String units,
        short usage,
        ArmID id);
    public ArmMetricNumericId64Definition
        newArmMetricNumericId64Definition(
        ArmApplicationDefinition app,
        String name,
        String units,
        short usage,
        ArmID id);
    public ArmMetricString32Definition newArmMetricString32Definition(
        ArmApplicationDefinition app,
        String name,
        String units,
        short usage,
        ArmID id);

    /* --- other definitions --- */
    public ArmMetricGroupDefinition newArmMetricGroupDefinition(
        ArmMetricDefinition[] definitions);
    public ArmTransactionWithMetricsDefinition
        newArmTransactionWithMetricsDefinition(
        ArmApplicationDefinition app,
        String name,
        ArmIdentityPropertiesTransaction identityProperties,
        ArmMetricGroupDefinition definition,
        ArmID id);

    /* --- metric instances --- */
    public ArmMetricCounter32 newArmMetricCounter32(
        ArmMetricCounter32Definition definition);
    public ArmMetricCounter64 newArmMetricCounter64(
        ArmMetricCounter64Definition definition);
    public ArmMetricCounterFloat32 newArmMetricCounterFloat32(
        ArmMetricCounterFloat32Definition definition);
    public ArmMetricGauge32 newArmMetricGauge32(
        ArmMetricGauge32Definition definition);
    public ArmMetricGauge64 newArmMetricGauge64(
```

```
        ArmMetricGauge64Definition definition);
    public ArmMetricGaugeFloat32 newArmMetricGaugeFloat32(
        ArmMetricGaugeFloat32Definition definition);
    public ArmMetricNumericId32 newArmMetricNumericId32(
        ArmMetricNumericId32Definition definition);
    public ArmMetricNumericId64 newArmMetricNumericId64(
        ArmMetricNumericId64Definition definition);
    public ArmMetricString32 newArmMetricString32(
        ArmMetricString32Definition definition);
    /* --- other interfaces --- */
    public ArmMetricGroup newArmMetricGroup(
        ArmMetricGroupDefinition groupDefinition,
        ArmMetric[] metrics);
    public ArmTranReportWithMetrics newArmTranReportWithMetrics(
        ArmApplication app,
        ArmTransactionWithMetricsDefinition definition,
        ArmMetricGroup group);
    public ArmTransactionWithMetrics newArmTransactionWithMetrics(
        ArmApplication app,
        ArmTransactionWithMetricsDefinition definition,
        ArmMetricGroup group);
    public boolean setErrorCallback(ArmErrorCallback errorCallback);
}
```

12.30 org.opengroup.arm40.metric.ArmMetricGauge32

ArmMetricGauge32 implements a 32-bit integer gauge. It is the same as ARM 2.0 metric type=4 (ARM_Gauge32).

```
public interface ArmMetricGauge32 extends ArmMetric {
// No Public Constructors
// Public Instance Methods
    public int get();
    public int set(int value);
}
```

12.31 org.opengroup.arm40.metric.ArmMetricGauge32Definition

ArmMetricGauge32Definition is a subclass of ArmMetricDefinition and serves as a marker interface that binds the metadata in ArmMetricDefinition to the metric data type, and describes an object interface that can be instantiated (there are no factory methods for objects that implement only ArmMetricDefinition). No new methods beyond those in ArmMetricDefinition are added.

```
public interface ArmMetricGauge32Definition extends
    ArmMetricDefinition {
// No Public Constructors
// No new Public Instance Methods.
// All methods are inherited from ArmMetricDefinition.
}
```

12.32 org.opengroup.arm40.metric.ArmMetricGauge64

ArmMetricGauge64 implements a 64-bit integer gauge. It is the same as ARM 2.0 metric type=5 (ARM_Gauge64).

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

12.33 org.opengroup.arm40.metric.ArmMetricGauge64Definition

ArmMetricGauge64Definition is a subclass of ArmMetricDefinition and serves as a marker interface that binds the metadata in ArmMetricDefinition to the metric data type, and describes an object interface that can be instantiated (there are no factory methods for objects that implement only ArmMetricDefinition). No new methods beyond those in ArmMetricDefinition are added.

```
public interface ArmMetricGauge64Definition extends
    ArmMetricDefinition {
// No Public Constructors
// No new Public Instance Methods.
// All methods are inherited from ArmMetricDefinition.
}
```

12.34 org.opengroup.arm40.metric.ArmMetricGaugeFloat32

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 the C bindings for ARM 2.0 and ARM 4.0, a floating-point value is provided directly. This was not done with the C bindings because ARM would have to support multiple floating-point formats, depending on the programming language and/or machine architecture, and the complexity was not deemed worthwhile.

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

12.35 org.opengroup.arm40.metric.ArmMetricGaugeFloat32Definition

ArmMetricGaugeFloat32Definition is a subclass of ArmMetricDefinition and serves as a marker interface that binds the metadata in ArmMetricDefinition to the metric data type, and describes an object interface that can be instantiated (there are no factory methods for objects that implement only ArmMetricDefinition). No new methods beyond those in ArmMetricDefinition are added.

```
public interface ArmMetricGaugeFloat32Definition extends
    ArmMetricDefinition {
// No Public Constructors
// No new Public Instance Methods.
// All methods are inherited from ArmMetricDefinition.
}
```

12.36 org.opengroup.arm40.metric.ArmMetricGroup

ArmMetricGroup is used to bind objects that implement a subclass of ArmMetric to an ArmTransactionWithMetrics or ArmTranReportWithMetrics object. The binding occurs when the transaction object is created [using **newArmTransactionWithMetrics()** or **newArmTranReportWithMetrics()**] and is immutable afterwards.

index is the index into the ArmMetric array. It must have a value in the range 0:6. To remain consistent with ARM 2.0, any ArmMetric subclass except ArmMetricString32 can be assigned to elements 0:5 and only ArmMetricString32 can be assigned to element 6.

getDefinition()
Returns the metric group definition used to create this object.

getMetric(index**)**
Returns the metric at the array index. This value may be null.

isMetricValid(index**)**
Indicates whether an ArmMetric subclass at this array index is valid.

setMetricValid(index**)**
Indicates whether an ArmMetric subclass at this array index is valid when any of the following calls are made. If the valid flag is set, then the metric value is processed.

— ArmTranReportWithMetrics: **report()**

— ArmTransactionWithMetrics: **start()**, **update()**, **stop()**

```
public interface ArmMetricGroup extends ArmInterface {
// No Public Constructors
// Public Instance Methods
    public ArmMetricGroupDefinition getDefinition();
    public ArmMetric getMetric(int index);
    public boolean isMetricValid(int index);
    public int setMetricValid(int index, boolean value);
}
```

12.37 org.opengroup.arm40.metric.ArmMetricGroupDefinition

ArmMetricGroupDefinition is used to bind ArmMetricDefinition objects to an ArmTransactionWithMetricsDefinition or ArmTranReportWithMetricsDefinition object. The binding occurs when the transaction object is created using **newArmTransactionWithMetricsDefinition()** or **newArmTranReportWithMetricsDefinition()** and is immutable afterwards.

index is the index into the ArmMetricDefinition array. It must have a value in the range 0:6. To remain consistent with ARM 2.0, any ArmMetricDefinition subclass except ArmMetricString32Definition can be assigned to elements 0:5 and only ArmMetricString32Definition can be assigned to element 6.

getMetricDefinition(index**)** returns the metric definition at the array index. This value may be null.

```
public interface ArmMetricGroupDefinition extends ArmInterface {
// No Public Constructors
// Public Instance Methods
    public ArmMetricDefinition getMetricDefinition(int index);
}
```

12.38 org.opengroup.arm40.metric.ArmMetricNumericId32

ArmMetricNumericId32 implements a 32-bit integer numeric ID. It is the same as ARM 2.0 metric type=7 (ARM_NumericID32).

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

12.39 org.opengroup.arm40.metric.ArmMetricNumericId32Definition

ArmMetricNumericId32Definition is a subclass of ArmMetricDefinition and serves as a marker interface that binds the metadata in ArmMetricDefinition to the metric data type, and describes an object interface that can be instantiated (there are no factory methods for objects that implement only ArmMetricDefinition). No new methods beyond those in ArmMetricDefinition are added.

```
public interface ArmMetricNumericId32Definition extends
    ArmMetricDefinition {
// No Public Constructors
// No new Public Instance Methods.
// All methods are inherited from ArmMetricDefinition.
}
```

12.40 org.opengroup.arm40.metric.ArmMetricNumericId64

ArmMetricNumericId64 implements a 64-bit integer numeric ID. It is the same as ARM 2.0 metric type=8 (ARM_NumericID64).

```
public interface ArmMetricNumericId64 extends ArmMetric {
// No Public Constructors
// Public Instance Methods
    public long get();
    public int set(long value);
}
```

12.41 org.opengroup.arm40.metric.ArmMetricNumericId64Definition

ArmMetricNumericId64Definition is a subclass of ArmMetricDefinition and serves as a marker interface that binds the metadata in ArmMetricDefinition to the metric data type, and describes an object interface that can be instantiated (there are no factory methods for objects that implement only ArmMetricDefinition). No new methods beyond those in ArmMetricDefinition are added.

```
public interface ArmMetricNumericId64Definition extends
    ArmMetricDefinition {
// No Public Constructors
// No new Public Instance Methods.
// All methods are inherited from ArmMetricDefinition.
}
```

12.42 org.opengroup.arm40.metric.ArmMetricString32

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.
- The limit of 32 in the ARM 2.0 C language interface is a *byte* limit. The limit in the ARM 4.0 C bindings is a *character* limit. Because a character may be represented by more than one byte (e.g., a character in UTF-8 is represented as 1, 2, or 3 bytes), the ARM 4.0 C metric may be longer than 32 bytes.

```
public interface ArmMetricString32 extends ArmMetric {
// No Public Constructors
// Public Instance Methods
    public String get();
    public int set(String s);
}
```

12.43 org.opengroup.arm40.metric.ArmMetricString32Definition

ArmMetricString32Definition is a subclass of ArmMetricDefinition and serves as a marker interface that binds the metadata in ArmMetricDefinition to the metric data type, and describes an object interface that can be instantiated (there are no factory methods for objects that implement only ArmMetricDefinition). No new methods beyond those in ArmMetricDefinition are added.

```
public interface ArmMetricString32Definition extends
    ArmMetricDefinition {
// No Public Constructors
// No new Public Instance Methods.
// All methods are inherited from ArmMetricDefinition.
}
```

12.44 org.opengroup.arm40.transaction.ArmPrestartTimeStats

[New in ARM 4.1.]

ArmPrestartTimeStats contains the arithmetic mean of the preparation time measured over several transactions. Optionally, it may also contain the standard deviation, a count of transactions, and the interval duration over which the measurements were made.

ArmPrestartTimeStats is used in situations in which the context of a transaction is not known when the transaction begins to execute, and for which there is a non-trivial delay before the context is known. ARM requires that the full context of a transaction be known when ArmTransaction's **start()** is executed (because the correlator is generated at this time). In ARM 2.0 and 3.0 there is no way to capture any time spent processing the transaction before the context is known. ARM 4.0 introduced the concept of an "arrival time". The arrival time is when processing of the transaction commenced. By default it is the moment in time when **start()** executes. If the delay between the start of processing and the execution of **start()** is significant, the application can measure the time and set the value in ArmTransaction before **start()** executes using **setPrestartTimeValue()**. The parameter to **setPrestartTimeValue()** is one of three types: a count in nanoseconds, a timestamp when the processing began, or the measured mean using ArmPrestartTimeStats.

There are getter and setter methods for each of four variables:

- **count()** is the current count of transactions used to calculate the mean. Note that this is *not* a rolling counter. It is a gauge whose value could increase or decrease over each interval. Its use is optional. A value of zero indicates that the count is not being set.
- **intervalMillis()** is the current interval duration in milliseconds over which transactions were used to calculate the mean. Note that this is *not* a rolling counter. It is a gauge whose value could increase or decrease over each interval. Its use is optional. A value of zero indicates that the interval is not being set.
- **meanNanos()** is the arithmetic mean in nanoseconds of the preparation time. A value of zero is ignored, and in this situation, all the other variables are also ignored.
- **standardDeviationNanos()** is the standard deviation in nanoseconds of the preparation time. A value of zero is ignored.

```
public interface ArmPrestartTimeStats extends ArmInterface {
// New in ARM 4.1
// No Public Constructors
// Public Instance Methods
    public int  getCount();
    public int  getIntervalMillis();
    public long getMeanNanos();
    public long getStandardDeviationNanos();
    public int  setCount(int count);
    public int  setIntervalMillis(int intervalMillis);
    public int  setMeanNanos(long meanNanos);
    public int  setStandardDeviationNanos(long standardDeviationNanos);
}
```

12.45 org.opengroup.arm40.transaction.ArmProperties

[New in ARM 4.1.]

ArmProperties is new in ARM 4.1. In ARM 4.1 it is subclassed by ArmDiagnosticProperties to provide a means to pass a set of (name,value) pairs when a transaction stops. It differs from ArmIdentityProperties primarily in that both the name and value parameters are mutable, and there is no concept of the properties being arrays. This interface is similar to the java.util.Properties class.

The property names can be any string with a maximum of 127 characters, with one restriction. Strings beginning with the four characters "ARM:" are reserved for the ARM specification and may only be used if they adhere to semantics defined in the specification. One name format is currently defined. Any name beginning with the eight-character prefix "ARM:CIM:" represents a name defined using the DMTF CIM (Distributed Management Task Force Common Information Model) naming rules. For example, "ARM:CIM:CIM_SoftwareElement.Name" indicates that the property value has the semantics of the Name property of the CIM_SoftwareElement class. It is anticipated that additional naming semantics are likely to be added in the future.

At present there is no factory method to instantiate an ArmProperties directly. There are methods to instantiate the ArmDiagnosticProperty subclass.

Interface methods:

- **setProperty()** sets the name and value of a property. Setting a value of null is equivalent to eliminating the property name from the set. The returned value is negative if an error occurs. This could occur if an ArmProperties subclass has some constraints, such as the maximum number of properties or the maximum length of a property name or value that would be violated by the set operation.
- **getProperty()** returns the value of a specified property, or null if the property name does not exist or the property value is set to null.
- **clearProperties()** is equivalent to executing **setProperty(**String name, null**)** for every property in the set. The returned value will be negative if an error occurs for some as yet unknown reason.

```
public interface ArmProperties extends ArmInterface {
// New in ARM 4.1
// No Public Constructors
// Public Instance Methods
    public int    clearProperties();
    public String getProperty(String name);
    public int    setProperty(String name, String value);
}
```

12.46 org.opengroup.arm40.tranreport.ArmSystemAddress

ArmSystemAddress encapsulates the network addressing information for a system. It may be used with ArmTranReport if the reported transaction executed on a different system.

- Address. The system address 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.
- Format. The format of the system address, such as an SNA address or a hostname.
- ID. An optional 16-byte ID associated with the format and address, if any. The ID value is bound to a unique combination of the format and address. When provided, the ID may be used as a concise alias for the unique combination. It may be null.

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 ArmSystemAddress.
- 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.

getAddress() returns a byte array containing the address. The returned value is the same value passed to the **newArmSystemAddress()** method of ArmTranReportFactory.

getFormat() returns a short containing the format. The returned value is the same value passed to the **newArmSystemAddress()** method of ArmTranReportFactory.

getID() returns the optional 16-byte ID associated with the format and address, if any. The returned value, which could be null, is the same value passed to the **newArmSystemAddress()** method of ArmTranReportFactory.

The fields are set using the **newArmSystemAddress()** method of ArmTranReportFactory or the **getArmSystemAddress()** method of ArmSystem. There are no setter methods for the individual fields. The object is immutable.

```
public interface ArmSystemAddress 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_X25;
    public static final short FORMAT_UUID;
// 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 byte[] getAddress();
    public short getFormat();
    public ArmID getID();
}
```

12.47 org.opengroup.arm40.transaction.ArmTimestamp

[New in ARM 4.1.]

ArmTimestamp is an abstract interface which is a superclass of specific timestamp formats: ArmTimestampOpaque, ArmTimestampStrings, or ArmTimestampUsecJan1970. ArmTimestamp is abstract in the sense that any ArmTimestamp object returned by any method in this specification satisfies one of the subclass interfaces.

The ArmTimestamp subclasses are used in situations in which the context of a transaction is not known when the transaction begins to execute, and for which there is a non-trivial delay before the context is known. ARM requires that the full context of a transaction be known when ArmTransaction's **start()** is executed (because the correlator is generated at this time). In ARM 2.0 and 3.0 there is no way to capture any time spent processing the transaction before the context is known. ARM 4.0 introduced the concept of an "arrival time". The arrival time is when processing of the transaction commenced. By default it is the moment in time when **start()** executes. If the delay between the start of processing and the execution of **start()** is significant, the application can capture the arrival time by creating an ArmTimestamp, setting the timestamp using **set()**, then associating it with an ArmTransaction using ArmTransaction's **setPrestartTimeValue(**ArmTimestamp**)** prior to calling ArmTransaction's **start()**.

The following considerations influence the selection of which ArmTimestamp subclass to use:

- If the application can invoke ARM interfaces at the moment that processing of the transaction begins, use ArmTimestampOpaque. Calling ArmTimestampOpaque's **set()** stores a timestamp with the current time.

  Note: The same result can be achieved by invoking ArmTransaction's **setArrivalTime()**. The use of ArmTimestampOpaque provides more flexibility and is preferred. Beginning with ARM 4.1, **setArrivalTime()** is deprecated in favor of using ArmTimestamp, but **setArrivalTime()** must be supported by all implementations of ARM 4.x.

- If the application cannot invoke ARM interfaces at the moment that processing of the transaction begins, but can capture the current time through some other means, use either ArmTimestampStrings or ArmTimestampUsecJan1970, converting the captured time into the appropriate format.

```
public interface ArmTimestamp extends ArmInterface {
// New in ARM 4.1
// No Public Constructors (also no factory method to create)
// No Public Instance Methods
}
```

12.48 org.opengroup.arm40.transaction.ArmTimestampOpaque

[New in ARM 4.1.]

ArmTimestampOpaque represents a timestamp in a format opaque to the application. ArmTimestampOpaque may be used with ArmTransaction's **setPrestartTimeValue(**ArmTimestamp**)** to provide the time when processing of a transaction began. See the discussion in Section 12.47 for more usage information.

Using ArmTimestampOpaque with ArmTransaction's **setPrestartTimeValue(**ArmTimestamp**)** is equivalent to but preferred to using ArmTransaction's **setArrivalTime()**.

set() stores the current time within the object.

There is no **get()** method because the format is unknown, so there is no point in getting the value.

```
public interface ArmTimestampOpaque extends ArmTimestamp {
// New in ARM 4.1
// No Public Constructors
// Public Instance Methods
    public int set();
}
```

12.49 org.opengroup.arm40.transaction.ArmTimestampStrings

[New in ARM 4.1.]

ArmTimestampStrings represents a timestamp in a format using character strings. ArmTimestampStrings may be used with ArmTransaction's **setPrestartTimeValue(**ArmTimestamp**)** to provide the time when processing of a transaction began. See the discussion in Section 12.47 for more usage information.

set() stores the current date and time within the object. The three parameters are:

- yyyymmdd = The year, month, and day, all in characters representing decimal digits. The pointer must not be null.
- hhmmssth = The hours, minutes, seconds, tenths, and hundredths of seconds, all in characters representing decimal digits. The pointer must not be null.
- muuu = The milliseconds and microseconds, all in characters representing decimal digits. A null value is equivalent to setting the values to all zeros.

getDate() returns the date portion of the current time.

getTime() returns the hours, minutes, seconds, tenths, and hundredths of seconds of the current time.

getTimeUsec() returns the milliseconds and microseconds of the current time.

```
public interface ArmTimestampString extends ArmTimestamp {
// New in ARM 4.1
// No Public Constructors
// Public Instance Methods
    public String getDate();
    public String getTime();
    public String getTimeUsec();
    public int set(String yyyymmdd, String hhmmssth, String muuu);
}
```

12.50 org.opengroup.arm40.transaction.ArmTimestampUsecJan1970

[New in ARM 4.1.]

ArmTimestampUsecJan1970 represents a timestamp as the number of microseconds since Jan 1, 1970 UTC. This is the same as a common Java format, except the resolution is microseconds instead of milliseconds.

ArmTimestampUsecJan1970 may be used with ArmTransaction's **setPrestartTimeValue(**ArmTimestamp**)** to provide the time when processing of a transaction began. See the discussion in Section 12.47 for more usage information.

set() stores the number of microseconds since Jan 1, 1970 UTC within the object. The returned value is negative if an error occurred.

get() returns the current value.

```
public interface ArmTimestampUsecJan1970 extends ArmTimestamp {
// New in ARM 4.1
// No Public Constructors
// Public Instance Methods
    public long get();
    public int  set(long microseconds);
}
```

12.51 org.opengroup.arm40.transaction.ArmToken

ArmToken is an abstract interface which is a superclass of ArmCorrelator, ArmSystemAddress (in the tranreport package), and ArmID, 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. The data token is always immutable. Subclasses of ArmToken can be thought of as wrappers around the data token.

The methods are as follows:

- **copyBytes()** is used to copy the token to a byte array that is already allocated. There are two forms. Both return a boolean. If the return value is true, the operation was successful. If the return value is false, the operation was not successful and the contents of the target array are undetermined. The most likely errors are an attempt to copy into a null pointer or into an array that is not long enough to hold the entire token.

  — **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 into which the token is copied. It is equivalent to creating a byte array of length **getLength()** and then executing **copyBytes()** into the new array. The ARM implementation would typically not keep a reference to the array, because that would interfere with garbage collection.

- **getLength()** returns the size of the byte array part of the token.

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 overrides the java.lang.Object methods **equals(**Object**)** and **hashCode()**. The behavior of these methods must be as follows.

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, ArmSystemAddress, or ArmID.

- **a.getBytes()** and **b.getBytes()** would return byte arrays of identical lengths and contents.

- If a subclass of ArmToken defines other data values (specifically, ArmSystemAddress 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-defined. 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.

```
public interface ArmToken extends ArmInterface {
// 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();
}
```

12.52 org.opengroup.arm40.tranreport.ArmTranReport

ArmTranReport is similar to ArmTransaction. Both are used to provide data about executing transactions. Instances of both are created based on metadata represented by an ArmTransactionDefinition, which in turn is scoped by an application definition. Both are scoped by a running application instance, represented by ArmApplication. There are two fundamental differences:

- With ArmTransaction, the response time is measured based on **start()** and **stop()** events. With ArmTranReport, the application measures the response time, and reports it with a single **report()** event.
- With ArmTransaction, the transaction always executes on the local system in the same JVM (Java Virtual Machine). With ArmTranReport, the transaction may execute in the same JVM, in a different JVM on the same system, or on a different system.

When executing in the same JVM, the ArmTranReport object is created with an ArmApplication. When executing in a different JVM on the same system or on a different system, the ArmTranReport object is created with an ArmApplicationRemote (a subclass of ArmApplication).

The two key methods of ArmTranReport are **generateCorrelator()** and **report()**:

- **generateCorrelator()** generates a new correlator using the immutable data set in the factory method and the current property values set by the four setter methods. It is assumed that **generateCorrelator()** is executed zero or once per transaction. The practical ramification is that the method implementation will update its internal state to have a unique identifier for one transaction (the equivalent of ArmTransaction's start handle). **getCorrelator()** returns the most recently generated ArmCorrelator or null, if **generateCorrelator()** has never been executed.
- **report()** is used to provide measurements about a completed transaction. There are two forms, each of which may also have either a diagnostic properties or a diagnostic detail string. Both provide the status (one of the STATUS_* constants in ArmConstants) and the response time (in nanoseconds). One also provides a stop time in the form of milliseconds since January 1, 1970, which is the same format returned by **java.lang.System.currentTimeMillis()**. If a stop time is not provided, or a stop time of –1 (USE_CURRENT_TIME) is provided, the ARM implementation substitutes the current time; that is, the time when the **report()** method executes. The optional form that takes a string is a way for an application to provide additional diagnostic details when the status is something other than STATUS_GOOD.

As noted above, **generateCorrelator()** updates the internal state of a transaction. The first time **report()** executes after **generateCorrelator()**, **report()** will not update the internal state for the transaction; it will use the transaction identifier from the **generateCorrelator()**. If **report()** executes twice in succession, or if **generateCorrelator()** has never been executed, **report()** will update the internal state for the transaction.

Summarizing, there are two patterns:

- If correlators for the current transaction are not requested, **generateCorrelator()** is not used. **report()** is executed after each transaction completes, and each time it generates a new transaction identifier, like a start handle.
- If correlators for the current transaction are requested, **generateCorrelator()** and **report()** are used in pairs. First **generateCorrelator()** establishes the transaction identifiers, as well as creating a correlator. This correlator is sent to downstream transactions. After the downstream transactions complete, and the current transaction completes, **report()** provides the measurements. In this case **report()** does not update the transaction identifier.

In addition to the identity properties from ArmApplication and ArmTransactionDefinition, there are four optional setter methods to establish additional instance-level context. They can be used at any time to update the attribute within the object. The only time the properties are meaningful is when **generateCorrelator()** or **report()** executes. At the moment either method executes, the current values are used, any or all of which may be null.

- **setContextURIValue()** sets the URI context value. **getContextURIValue()** returns the value.
- **setContextValue()** sets one of the maximum 20 context properties that may change for each transaction. **getContextValue()** returns the value, whether null or not. The context property name at the specified array index must have been set to a non-null value when the ArmTransactionDefinition object was created. If the name is null, the value will be set to null.
- **setParentCorrelator()** sets the correlator of the parent transaction. **getParentCorrelator()** returns the value, whether null or not.
- **setUser()** associates a user, represented by an instance of ArmUser, to the ArmTranReport instance. This user is assumed to be the user for all **start()**/**stop()** pairs until the association is changed or cleared. **setUser(**null**)** clears any existing association to an ArmUser. **getArmUser()** returns the current value, whether null or not.

There are two methods that return the data used in the factory method to create this instance of ArmTranReport: **getApplication()** and **getDefinition()**.

```
public interface ArmTranReport extends ArmInterface {
// No Public Constructors
// Public Instance Methods
    public ArmCorrelator generateCorrelator();
    public ArmApplication getApplication();
    public String getContextURIValue();
    public String getContextValue(int index);
    public ArmCorrelator getCorrelator();
    public ArmCorrelator getParentCorrelator();
    public long getResponseTime();
    public int getStatus();
    public ArmTransactionDefinition getDefinition();
    public ArmUser getUser();
    public int report(int status, long respTimeNanos);
    public int report(int status, long respTimeNanos, long stopTime);
```

```
    public int report(int status, long respTimeNanos, String
        diagnosticDetail);
    public int report(int status, long respTimeNanos, long stopTime,
        String diagnosticDetail);
    public int report(int status, long respTimeNanos,
        ArmDiagnosticProperties props);                          //4.1
    public int report(int status, long respTimeNanos, long stopTime,
        ArmDiagnosticProperties props);                          //4.1
    public int setContextURIValue(String value);
    public int setContextValue(int index, String value);
    public int setParentCorrelator(ArmCorrelator parent);
    public int setUser(ArmUser user);
}
```

12.53 org.opengroup.arm40.tranreport.ArmTranReportFactory

ArmTranReportFactory provides methods to create instances of the classes in the org.opengroup.arm40.tranreport package.

Factory Version: The org.opengroup.arm40.tranreport package contains the interfaces defined in both ARM 4.0 and ARM 4.1. Applications that use ARM 4.1 could instantiate classes that implement ARM 4.0 but not ARM 4.1. See Section 8.2.1 for an explanation of how to handle this situation to avoid runtime errors.

newArmApplicationRemote()
: Creates an ArmApplicationRemote. See the ArmApplicationRemote description for details about the parameters. If systemAddress is null, the addressing information for the local system is used.

newArmSystemAddress()
: Creates an ArmSystemAddress from the specified format and the input byte array. See the ArmSystemAddress description for details about the parameters.

newArmTranReport()
: Creates an object that represents an executing transaction. The metadata is supplied in an ArmTransactionDefinition object. It is scoped by an application instance, represented by ArmApplication (or its subclass, ArmApplicationRemote).

Error Handling Philosophy

If an invalid set of parameters is passed to any method, such as an offset that extends beyond the end of an array, an object is returned that may contain dummy data. For example, a null byte[] addressBytes parameter might result in creating an object with an address of all 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, even if the data may be at least partially meaningless. The **getErrorCode()** method of each object can be used after it is created to test whether errors occurred. Refer to Chapter 9 for a more complete explanation.

ArmTranReportFactory also serves as the anchor point for an application-registered callback function. **setErrorCallback()** is used to register a callback that will be called if any method of an object created by this factory object sets the error code to a negative value. The error code is retrieved using **getErrorCode()**, defined in ArmInterface, the root for most interfaces in the ARM specification. The boolean returned by **setErrorCallback()** indicates whether the registration is accepted. If an ARM implementation does not support the callback function, it will return false. **setErrorCallback(**null**)** unregisters any previously registered callback. Note that due to timing conditions or specifics of the ARM implementation, a previously registered callback may continue to be called for an indeterminate length of time after **setErrorCallback(**null**)** is executed. For a broader discussion of error handling, refer to Chapter 9.

```
public interface ArmTranReportFactory extends ArmInterface {
// Public Constants
```

```
    public static final String propertyKey;
// Public Instance Methods
    public ArmApplicationRemote newArmApplicationRemote(
        ArmApplicationDefinition definition,
        String group, String instance,
        String[] contextValues,
        ArmSystemAddress systemAddress);
    public ArmSystemAddress newArmSystemAddress(
        short format,
        byte[] addressBytes,
        ArmID id);
    public ArmSystemAddress newArmSystemAddress(
        short format,
        byte[] addressBytes,
        int offset,
        ArmID id);
    public ArmSystemAddress newArmSystemAddress(
        short format,
        byte[] addressBytes,
        int offset,
        int length,
        ArmID id);
    public ArmTranReport newArmTranReport(
        ArmApplication app,
        ArmTransactionDefinition definition);
    public boolean setErrorCallback(ArmErrorCallback errorCallback);
}
```

12.54 org.opengroup.arm40.metric.ArmTranReportWithMetrics

ArmTranReportWithMetrics is a subclass of ArmTranReport that is used if the application wishes to use metrics. All the ArmTranReport semantics for using **report()** apply to ArmTranReportWithMetrics.

ArmTranReportWithMetrics extends ArmTranReport by adding methods to manipulate metrics. The ArmMetric subclass objects are bound to an ArmTranReportWithMetrics object when the ArmTranReportWithMetrics is created. This is done by specifying ArmMetricGroup in the **newArmTranReportWithMetrics()** method of ArmMetricFactory.

getTransactionWithMetricsDefinition()

Returns the ArmTransactionWithMetricsDefinition object that contains the metadata describing this transaction, including the metric definitions.

getMetricGroup()

Returns the ArmMetricGroup object that was bound when ArmTranReportWithMetrics is created. The returned value may be null.

```
public interface ArmTranReportWithMetrics extends ArmTranReport {
// No Public Constructors
// Public Instance Methods
    public ArmTransactionWithMetricsDefinition
        getTransactionWithMetricsDefinition();
    public ArmMetricGroup getMetricGroup();
}
```

12.55 org.opengroup.arm40.transaction.ArmTransaction

For most applications, ArmTransaction is the most important of all the ARM classes, and the most frequently used. Many applications operate only on ArmTransaction objects after some initialization (using ArmTransactionFactory, ArmApplicationDefinition, ArmApplication, and ArmTransactionDefinition). Instances of ArmTransaction represent transactions when they execute. A "transaction" is any unit of work that has a clearly understood beginning and ending point, and which begins and ends in the same JVM (Java Virtual Machine) instance. Examples include a remote procedure call, a database transaction, and a batch job. It is not necessary that an ARM transaction implements robust functions such as commit and rollback.

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 re-use. Another strategy is to create one instance of each type per thread, which eliminates the need to manage the pool, handle synchronization if the pool is depleted, etc.

ArmTransaction is created with the **newArmTransaction()** method of ArmTransactionFactory. The metadata common to all instances is contained in the ArmTransactionDefinition used to create the object. Each transaction is scoped by an application instance, represented by ArmApplication.

The most frequently (and often the only) used methods are **start()**, **getCorrelator()**, and **stop()**. A typical sequence is as follows:

- Just prior to executing a transaction, such as a remote procedure call, call **start()** to signal to ARM that the measurable transaction is beginning. **start()** causes ARM to capture the current time. If a correlation token was received when your program was invoked, pass it as a parameter on the **start()**.
- Just after executing **start()**, and just prior to executing the transaction, call **getCorrelator()** to get a correlation token that can be sent along with the other transaction parameters to the receiver. Not all programs use correlators, but their use is highly recommended, because without them, it is usually impossible to drill down to understand the components of a transaction, where they executed, what resources they used, etc.
- Execute the transaction (e.g., make the remote procedure call).
- As soon as it ends, call **stop()**, passing a status to indicate whether the transaction succeeded. **stop()** causes ARM to capture the current time. The response time is determined by calculating the duration between the **start()** and **stop()** events.

Following are details about each method. All methods that return an int are returning an error code that the application may but need not test. Refer to Chapter 9 for more information about handling errors.

- **start()** indicates when a transaction begins. Because the response time depends on when **start()** executes, it should execute as close to the actual start time as possible. After **start()**

executes, it should not be executed again until **reset()** or **stop()** is executed. If **start()** executes consecutively, the behavior is undefined.

There are four versions of **start()**, depending on whether a parent correlator is provided, and if one is provided, the format of the input data. The length of the correlator is in the first two bytes of the correlator byte array, with the bytes in network byte order. When the input is a byte array, the length of the array does not matter, as long as it is at least long enough to hold the correlator, based on the two-byte length field.

- After a **start()** there can be any number of **update()** calls until a **stop()**. If it is executed at any other time, it is ignored. The behavior of **update()** issued at any other time is undefined. **update()** is used for three purposes:

  — It serves as a heartbeat to show that a transaction is still executing. This is especially useful if the transaction is a long-running job.

  — When used with ArmTransactionWithMetrics, a subclass of ArmTransaction (in the org.opengroup.arm40.metrics package), any of the metric values can be provided with an **update()**.

  — It can be used to pass ArmMessageSentEvent and ArmMessageReceivedEvent.

- **stop()** indicates when a transaction ends and what the status of the transaction was. Because the response time depends on when **stop()** executes, it should execute as close to the actual stop time as possible. If **stop()** is erroneously issued when there is no transaction active [**start()** issued without a matching **stop()**], it is ignored. The status must be one of STATUS_ABORT, STATUS_FAILED, STATUS_GOOD, or STATUS_UNKNOWN (all defined in ArmConstants). The optional form that takes a string is a way for an application to provide additional diagnostic details when the status is something other than STATUS_GOOD.

- **reset()** can be executed at any time. If a transaction is currently executing [**start()** executed without a matching **stop()**], the current transaction is discarded and treated as if the **start()** never executed. If no transaction is currently executing, the state of the object is unchanged. If there is any doubt about the state of an object, **reset()** gets the object into a known state in which a **start()** may be executed. **reset()** clears the arrival time and the current correlator; it does not change traceRequested or any of the context URI, context values, or user.

- **getCorrelator()** returns a reference to the correlator for the current transaction. It may be a newly created object. It can be executed anytime after **start()** is executed. Each time it is executed, it will return the same value until the next **stop()** or **reset()** is executed. If it is executed at any other time, it will return an ArmCorrelator object, but the data within the ArmCorrelator object is undefined and should not be used.

  getCorrelator(true) indicates that the application is requesting a correlator that will not be sent outside the current JVM instance. An ARM implementation may be able to optimize correlator handling if it knows this. **getCorrelator(false)** is equivalent to **getCorrelator()**.

- **setTraceRequested()** is used to suggest or withdraw a suggestion from an application that a transaction be traced. **isTraceRequested()** is used to query the current trace request state. The initial state is false. Once set, it remains in that state until set to a different state.

- **bindThread()** and **unbindThread()** can be called from any thread to indicate that the thread is executing on behalf of the transaction. This is useful when multiple threads execute the same logical (ARM) transaction, because instrumentation of resource consumption at the thread level can be more precise. The thread remains bound to this transaction until **unbindThread()** is executed in this thread or **stop()** or **reset()** is executed.

- **setAutomaticBindThread()** indicates that a thread is executing on behalf of a transaction at the moment **start()** executes. It is used in place of **bindThread()**. The motivation is to avoid needing to make another method call to **bindThread()** in the common case where each transaction executes as a separate thread. **setAutomaticBindThread(**false**)** resets this behavior.

- **blocked()** is used to indicate that the transaction is blocked waiting on an external transaction (which may or may not be instrumented with ARM) or some other event to complete. It has been found useful to separate out this "blocked" time from the elapsed time between the **start()** and **stop()**. **unblocked()** indicates when the blocking condition has ended. A transaction may be blocked by multiple conditions simultaneously. A "block handle" returned by **blocked()** is the input parameter to **unblocked()** to indicate which blocking condition has ended. Starting with ARM 4.1, an ArmBlockCause may be provided on the **blocked()** call to describe the cause of the block.

- [Deprecated in ARM 4.1 – the preferred mechanism is to use **setPrestartTimeValue(**ArmTimestamp**)**]
  setArrivalTime() can be used in situations in which the context of a transaction is not known when the transaction begins to execute, and for which there is a non-trivial delay before the context is known. ARM requires that the full context of a transaction be known when **start()** is executed (because the correlator is generated at this time). In ARM 2.0 and 3.0 there is no way to capture any time spent processing the transaction before the context is known. ARM 4.0 introduces the concept of an "arrival time". The arrival time is when processing of the transaction commenced, or when the transaction is available to be processed, such as waiting in a queue; it is left to the application developer's discretion to select the appropriate arrival time. By default, it is the moment in time when **start()** executes. If the delay between the start of processing and the execution of **start()** is significant, the application can capture the arrival time by invoking **setArrivalTime()**. This establishes a timestamp that will be used at the next **start()**, after which the value will be reset within the ArmTransaction object. The **reset()** and **stop()** methods also clear the value.

- [ARM 4.1] **setPrestartTimeValue()** is used in situations in which the context of a transaction is not known when the transaction begins to execute, and for which there is a non-trivial delay before the context is known. ARM requires that the full context of a transaction be known when ArmTransaction's **start()** is executed (because the correlator is generated at this time). In ARM 2.0 and 3.0 there is no way to capture any time spent processing the transaction before the context is known. ARM 4.0 introduced the concept of an "arrival time". The arrival time is when processing of the transaction commenced. By default it is the moment in time when **start()** executes. If the delay between the start of processing and the execution of **start()** is significant, the application can measure the time and set the value in ArmTransaction before **start()** executes using **setPrestartTimeValue()**. The parameter to **setPrestartTimeValue()** is one of three types:

a count in nanoseconds, a timestamp when the processing began (ArmTimestamp), or the measured mean using ArmPrestartTimeStats. No permanent association is made between ArmTransaction and either ArmTimestamp or ArmPrestartTimeStats – it is a copy by value into ArmTransaction. The copied value is valid for one execution of **start()**; it will be discarded after **start()** and must be set again to apply to a later **start()**.

- **setContextURIValue()** sets the URI context value. **getContextURIValue()** returns the value. In most scenarios, a URI would be used as a transaction identity property or a context property, but not both. The only allowed exception is when the base part of the URI is used as an identity property, and the full URI (e.g., with the parameters) is used as a context property. Any other use of URIs as both identity and context properties is invalid.

- **setContextValue()** sets one of the maximum 20 context properties that may change for each executing transaction. **getContextValue()** returns the value. The "name" part is available via **getDefinition().getIdentityProperties().getContextName()**. The values are position-sensitive – they match the position in the referenced context name array (see the discussion of ArmIdentityProperties for more details). The context property name at the specified array index must have been set to a non-null value when the ArmTransactionDefinition object was created. If the name is null or a zero-length string, both the name and value are ignored. If the value is null or a zero-length string, the meaning is that there is no value for this transaction. The value should not contain trailing blank characters or consist of only blank characters.

- **setUser()** associates a user, represented by an instance of ArmUser, to the ArmTransaction instance. This user is assumed to be the user for all **start()**/**stop()** pairs until the association is changed or cleared. **setUser(**null**)** clears any existing association to an ArmUser. **getUser()** returns the last value that was set.

- **getParentCorrelator()** returns the last value set on a **start()** method. If no value was set on the **start()** method, or if **start()** has never executed, it returns null.

- **getStatus()** returns the last value set on a **stop()** method. If **stop()** has never executed, it returns STATUS_INVALID.

- **getApplication()** returns the value passed to the **newArmTransaction()** method of ArmTransactionFactory.

- **getDefinition()** returns the value passed to the **newArmTransaction()** method of ArmTransactionFactory.

- **getControl()** may be used by applications to understand the ARM implementation's preferences for instrumentation details that apply at the scope of a single transaction. The getter methods in the returned control block, if any, may be queried to learn the details. If null is returned, the application should use its default or currently set level of instrumentation. **getControl()** would be used after the parent correlator, if any, is received and before the transaction starts processing.

- **setMessageEventGroup()** is used to pass an ArmMessageEventGroup to ARM. It is executed before **start(), update()**, **stop()**, **blocked()**, or **unblocked()**. During the execution of any of these five methods, the ArmMessageEventGroup is processed. After

any of these five methods execute, any reference to an ArmMessageEventGroup is set to null, though no change is made to the ArmMessageEventGroup.

setMessageEventGroup() must be executed again prior to one of the five listed methods to indicate that the ArmMessageEventGroup should be processed again.

setMessageEventGroup(null**)** clears any previously setting.

```
public interface ArmTransaction extends ArmInterface {
// No Public Constructors
// Public Instance Methods
    public int bindThread();
    public long blocked();
    public long blocked(ArmBlockCause cause);                    //4.1
    public ArmApplication getApplication();
    public String getContextURIValue();
    public String getContextValue(int index);
    public ArmCorrelator getCorrelator();
    public ArmCorrelator getCorrelator(boolean localOnly);       //4.1
    public ArmTransactionDefinition getDefinition();
    public ArmCorrelator getParentCorrelator();
    public int getStatus();
    public ArmTransactionControl getControl(ArmCorrelator parent);//4.1
    public ArmUser getUser();
    public boolean isAutomaticBindThread();                      //4.1
    public boolean isTraceRequested();
    public int reset();
    public int setArrivalTime();                        //DEPRECATED:4.1
    public int setAutomaticBindThread(boolean b);                //4.1
    public int setContextURIValue(String value);
    public int setContextValue(int index, String value);
    public int setMessageEventGroup(ArmMessageEventGroup group);  //4.1
    public int setPrestartTimeValue(long nanos);                 //4.1
    public int setPrestartTimeValue(ArmPrestartTimeStats stats);  //4.1
    public int setPrestartTimeValue(ArmTimestamp timestamp);     //4.1
    public int setTraceRequested(boolean traceState);
    public int setUser(ArmUser user);
    public int start();
    public int start(byte[] parentCorr);
    public int start(byte[] parentCorr, int offset);
    public int start(ArmCorrelator parentCorr);
    public int stop(int status);
    public int stop(int status, String diagnosticDetail);
    public int stop(int status, ArmDiagnosticProperties props);   //4.1
    public int unbindThread();
    public int unblocked(long blockHandle);
    public int update();
}
```

12.56 org.opengroup.arm40.transaction.ArmTransactionControl

[New in ARM 4.1]

ArmTransactionControl is used by applications to request the type and scope of instrumentation the ARM implementation prefers for a single transaction. Its use is optional for both applications and ARM implementations. Further, the control settings represent preferences; they are not binding on the application.

The scope of the settings applies to one executing transaction.

ArmTransactionDefinitionControl is created using ArmTransaction's **getControl()** method. Once created, the values are immutable.

Refer to the description of ArmApplicationControl for a description of all the methods and their meaning. Note that ArmTransactionControl, like ArmTransactionDefinitionControl, does not use four of ArmApplicationControl's methods:

- **isTransactionDefinitionControlUsed()**
- **isTransactionControlUsed()**
- **isPrivateDataRequested()**
- **isSecureDataRequested()**

```
public interface ArmTransactionControl extends ArmInterface {
// New in ARM 4.1
// No Public Constructors
// Public Instance Methods
    public int     getCollectionDepth();
    public boolean isBindThreadRequested();
    public boolean isBlockRequested();
    public boolean isDiagnosticDataRequested();
    public boolean isMessageEventDataRequested();
    public boolean isMetricDataRequested();
    public boolean isUserDataRequested();
}
```

12.57 org.opengroup.arm40.transaction.ArmTransactionDefinition

ArmTransactionDefinition contains the metadata that is the same for all instances of a transaction type (represented by ArmTransaction or ArmTranReport). It is created with the **newArmTransactionDefinition()** method of ArmTransactionFactory. ArmTransactionDefinition has the following attributes, all of which are immutable:

- The definition of the application that contains the transaction. It must not be null.
- The name of the transaction (maximum 127 characters).
- (optional) Identity property names and values and context property names in arrays. See the discussion of identity and context property names in ArmIdentityProperties.
- (optional) ID. An optional 16-byte ID may be associated with the identity of a transaction definition. The returned value, which could be null, is the same value passed to the **newArmTransactionDefinition()** method of ArmTransactionFactory. The ID value is bound to a unique combination of the application identity (represented by ArmApplicationDefinition), transaction name, any URI identity property, any identity property names and values, and any context property names. When provided, the ID may be used as a concise alias for the unique combination. It may be null.

```
public interface ArmTransactionDefinition extends ArmInterface {
// No Public Constructors
// Public Instance Methods
    public ArmApplicationDefinition getApplicationDefinition();
    public ArmID getID();
    public ArmIdentityPropertiesTransaction getIdentityProperties();
    public String getName();
}
```

12.58 org.opengroup.arm40.transaction.ArmTransactionFactory

The class that implements the ArmTransactionFactory interface is used to create the objects that implement interfaces in the org.opengroup.arm40.transaction package. Refer to the interface descriptions for an explanation of the parameters.

Factory Version: The org.opengroup.arm40.transaction package contains the interfaces defined in both ARM 4.0 and ARM 4.1. Applications that use ARM 4.1 could instantiate classes that implement ARM 4.0 but not ARM 4.1. See Section 8.2.1 for an explanation of how to handle this situation to avoid runtime errors.

newArmApplication()

Creates the ArmApplication object to which transactions are related. definition is the only required non-null parameter.

newArmApplicationDefinition()

Creates the ArmApplicationDefinition object that describes the metadata about an application; that is, the descriptive data that is the same for all instances of the same application. name is the only required non-null parameter.

[ARM 4.1] **newArmBlockCause()**

Creates an ArmBlockCause. It takes no parameters – all its properties are set via setter methods.

newArmCorrelator()

Creates a correlator from a byte array in the correct format. No length field is passed to **newArmCorrelator()** because ARM requires that the length of the correlator be found in the first two bytes (in network byte order) of the byte array (either at corrBytes or corrBytes+offset). The correlator must be no longer than the value of the constant org.opengroup.arm40.transaction.ArmConstants.CORR_MAX_LENGTH. If the correlator is longer than the ARM implementation supports, an ArmCorrelator object will be created, but it may contain dummy data, at the discretion of the ARM implementation.

newArmID()

Creates the objects that contain an immutable 16-byte ID. Having 16-byte IDs accommodates some widely used IDs, such as the UUID (Universally Unique Identifier) defined in the DCE and IETF standards.

newArmIdentityProperties()

Creates an object that contains an immutable set of identity property names and values, and an immutable set of context property names. These properties describe identity properties that are common to all instances of an application or transaction.

The names and values are provided in arrays of strings. There can be up to twenty elements in each of the arrays. A null reference and a zero-length string are both treated as a null element. It is permissible to have null elements in the middle of the array. For example, it is permissible for the elements at indices 0 and 19 to be non-null and all the elements from indices 1 to 18 to be null. The arrays are position-sensitive.

The identity property name and value arrays should contain the same number of elements at the same array indices; for each non-null name or value, there should be a corresponding non-null value or name, respectively, at the same array index. For any array index, if either the name or the value is null, both the name and the value are ignored.

The context property names are provided in this object. The context values are provided in an ArmApplication, ArmTranReport, or ArmTransaction object. If the name at an array index is null, the corresponding value in those objects is ignored.

newArmIdentityPropertiesTransaction()
Creates an object that extends newArmIdentityProperties to also include a URI property, which is also immutable. See the **newArmIdentityProperties()** description for all other parameters.

[ARM 4.1] **newMessageEventGroup()**
Creates an ArmMessageEventGroup. It takes no parameters – all its properties are set via setter methods.

[ARM 4.1] **newMessageReceivedEvent()**
Creates an ArmMessageReceivedEvent. It takes no parameters – all its properties are set via setter methods.

[ARM 4.1] **newMessageSentEvent()**
Creates an ArmMessageSentEvent. It takes no parameters – all its properties are set via setter methods.

[ARM 4.1] **newArmPrestartTimeStats()**
Creates an ArmPrestartTimeStats. It takes no parameters – all its properties are set via setter methods.

[ARM 4.1] **newArmTimestampOpaque()**
Creates an ArmTimestampOpaque. It takes no parameters – all its properties are set via setter methods.

[ARM 4.1] **newArmTimestampStrings()**
Creates an ArmTimestampStrings. It takes no parameters – all its properties are set via setter methods.

[ARM 4.1] **newArmTimestampUsecJan1970()**
Creates an ArmTimestampUsecJan1970. It takes no parameters – all its properties are set via setter methods.

newArmTransaction()
Creates an object that represents a transaction.

newArmTransactionDefinition()
Creates an object that represents the metadata about a transaction.

newArmUser()

Creates an ArmUser object that represents the user who invoked (directly or indirectly) the transaction.

Error Handling Philosophy

If an invalid set of parameters is passed to any method, such as an offset that extends beyond the end of an array, an object is returned that may contain dummy data. For example, a null byte[] addressBytes parameter might result in creating an object with an address of all 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, even if the data may be at least partially meaningless. The **getErrorCode()** method of each object can be used after it is created to test whether errors occurred. Refer to Chapter 9 for a more complete explanation.

ArmTransactionFactory also serves as the anchor point for an application-registered callback function. **setErrorCallback()** is used to register a callback that will be called if any method of an object created by this factory object sets the error code to a negative value. The error code is retrieved using **getErrorCode()**, defined in ArmInterface, the root for most interfaces in the ARM specification. The boolean returned by **setErrorCallback()** indicates whether the registration is accepted. If an ARM implementation does not support the callback function, it will return false. **setErrorCallback(**null**)** unregisters any previously registered callback. Note that due to timing conditions or specifics of the ARM implementation, a previously registered callback may continue to be called for an indeterminate length of time after **setErrorCallback(**null**)** is executed. For a broader discussion of error handling, refer to Chapter 9.

```
public interface ArmTransactionFactory extends ArmInterface {
// Public Constants
    public static final String propertyKey;
    public static final String propertyKey41;                    //4.1
// Public Instance Methods
    public ArmApplication newArmApplication(
        ArmApplicationDefinition definition,
        String group,
        String instance,
        String[] contextValues);
    public ArmApplicationDefinition newArmApplicationDefinition(
        String name,
        ArmIdentityProperties identityProperties,
        ArmID id);
    public ArmBlockCause newArmBlockCause();                     //4.1
    public ArmCorrelator newArmCorrelator(
        byte[] corrBytes);
    public ArmCorrelator newArmCorrelator(
        byte[] corrBytes,
        int offset);
    public ArmID newArmID(
        byte[] idBytes);
    public ArmID newArmID(
        byte[] idBytes,
        int offset);
```

```
    public ArmIdentityProperties newArmIdentityProperties(
        String[] identityNames,
        String[] identityValues,
        String[] contextNames);
    public ArmIdentityPropertiesTransaction
        newArmIdentityPropertiesTransaction(
        String[] identityNames,
        String[] identityValues,
        String[] contextNames,
        String uriValue);
    public ArmMessageEventGroup newMessageEventGroup();           //4.1
    public ArmMessageReceivedEvent newMessageReceivedEvent();     //4.1
    public ArmMessageSentEvent newMessageSentEvent();             //4.1
    public ArmPrestartTimeStats newArmPrestartTimeStats();        //4.1
    public ArmTimestampOpaque newArmTimestampOpaque();            //4.1
    public ArmTimestampStrings newArmTimestampStrings();          //4.1
    public ArmTimestampUsecJan1970 newArmTimestampUsecJan1970();  //4.1
    public ArmTransaction newArmTransaction(
        ArmApplication app,
        ArmTransactionDefinition definition);
    public ArmTransactionDefinition newArmTransactionDefinition(
        ArmApplicationDefinition app,
        String name,
        ArmIdentityPropertiesTransaction identityProperties,
        ArmID id);
    public ArmUser newArmUser(
        String name,
        ArmID id);
    public boolean setErrorCallback(ArmErrorCallback errorCallback);
}
```

12.59 org.opengroup.arm40.transaction.ArmTransactionDefinitionControl

[New in ARM 4.1]

ArmTransactionDefinitionControl is used by applications to request the type and scope of instrumentation the ARM implementation prefers for a registered transaction ID. Its use is optional for both applications and ARM implementations. Further, the control settings represent preferences; they are not binding on the application.

The scope of the settings applies to all transactions with the registered transaction ID. Some of these settings may be overridden by ArmTransactionControl.

ArmTransactionDefinitionControl is created using ArmApplication's **getTransactionDefinitionControl()** method. Once created, the values are immutable. It is a good practice to periodically get a new ArmTransactionDefinitionControl, and discard the old one, to see if there have been any changes.

Refer to the description of ArmApplicationControl for a description of all the methods and their meaning. Note that ArmTransactionDefinitionControl does not use four of ArmApplicationControl's methods:

- **isTransactionDefinitionControlUsed()**
- **isTransactionControlUsed()**
- **isPrivateDataRequested()**
- **isSecureDataRequested()**

```
public interface ArmTransactionDefinitionControl extends ArmInterface {
// New in ARM 4.1
// No Public Constructors
// Public Instance Methods
    public int     getCollectionDepth();
    public boolean isBindThreadRequested();
    public boolean isBlockRequested();
    public boolean isDiagnosticDataRequested();
    public boolean isMessageEventDataRequested();
    public boolean isMetricDataRequested();
    public boolean isUserDataRequested();
}
```

12.60 org.opengroup.arm40.metric.ArmTransactionWithMetrics

ArmTransactionWithMetrics is a subclass of ArmTransaction which is used if the application wishes to use metrics. All the ArmTransaction rules for using **start()**, **stop()**, etc. apply to ArmTransactionWithMetrics.

ArmTransactionWithMetrics extends ArmTransaction by adding methods to manipulate metrics. 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.

getTransactionWithMetricsDefinition()

Returns the ArmTransactionWithMetricsDefinition object that contains the metadata describing this transaction, including the metric definitions.

getMetricGroup()

Returns the ArmMetricGroup object that was bound when ArmTransactionWithMetrics is created. The returned value may be null.

```
public interface ArmTransactionWithMetrics extends ArmTransaction {
// No Public Constructors
// Public Instance Methods
    public ArmTransactionWithMetricsDefinition
        getTransactionWithMetricsDefinition();
    public ArmMetricGroup getMetricGroup();
}
```

12.61 org.opengroup.arm40.metric.ArmTransactionWithMetricsDefinition

ArmTransactionWithMetricsDefinition subclasses ArmTransactionDefinition to add a binding with an ArmMetricGroupDefinition. It contains the metadata that is the same for all ArmTransactionWithMetrics (or ArmTranReportWithMetrics) instances with the same identity.

The properties that are accessible via this interface are the same as ArmTransactionDefinition, plus the following:

- The metric group definition through which the metric definitions are known. The metric definitions contain the metadata about the metrics.

```
public interface ArmTransactionWithMetricsDefinition extends
    ArmTransactionDefinition {
// No Public Constructors
// Public Instance Methods (in addition to several inherited from
// ArmTranscationDefinition)
    public ArmMetricGroupDefinition getMetricGroupDefinition();
}
```

12.62 org.opengroup.arm40.transaction.ArmUser

ArmUser represents a user on behalf of whom a transaction is executed. It is created with the **newArmUser()** method of ArmTransactionFactory. It has the following attributes, all of which are immutable:

- Name: The maximum length is 127 characters (CIM allows 256 but ARM 2.0 allows 128 bytes, including the null-termination character, so 127 is used). The name must not be null or zero-length.
- (optional) A 16-byte ID is optionally associated with each ArmUser. It is provided by the application. If the value is null, no ID was provided.

```
public interface ArmUser extends ArmInterface {
// No Public Constructors
// Public Instance Methods
    public ArmID getID();
    public String getName();
}
```

A Application Instrumentation Samples

A.1 Basic End-to-End Measurements

```
/* ------------------------------------------------------------------------ */
/* --------- ARM 4.1 Java API Example: using basic capabilities  ---------- */
/* ------------------------------------------------------------------------ */

package org.opengroup.arm40.examples;

import org.opengroup.arm40.transaction.ArmApplication;
import org.opengroup.arm40.transaction.ArmApplicationDefinition;
import org.opengroup.arm40.transaction.ArmConstants;
import org.opengroup.arm40.transaction.ArmCorrelator;
import org.opengroup.arm40.transaction.ArmTransaction;
import org.opengroup.arm40.transaction.ArmTransactionDefinition;
import org.opengroup.arm40.transaction.ArmTransactionFactory;

public class Arm40BasicExample {

   ArmTransactionFactory tranFactory;
   ArmApplicationDefinition armAppDef;
   ArmApplication armApp;
   ArmTransactionDefinition armParentTranDef;

   private ArmTransactionDefinition armChildTranDef;

   /* initialize factories, ARM definitions and an ARM application instance */
   void armRegisterAndInit() {
      String tranFactoryName = System.getProperties().getProperty(
         ArmTransactionFactory.propertyKey,
         "org.opengroup.arm40.sdk.ArmTransactionFactoryImpl");

      try {
         tranFactory = (ArmTransactionFactory)
            Class.forName(tranFactoryName).newInstance();
      } catch (Exception e) {
         // Note: this example does not provide error handling at all
         System.exit(-1);
      }

      //
      armAppDef = tranFactory.newArmApplicationDefinition(
         // provide a name, no properties or ID
         "ARM Java Spec Basic Example", null, null);

      armParentTranDef = tranFactory.newArmTransactionDefinition(
         // only provide a name, no properties or ID
         armAppDef, "JExampleParentTx", null, null);

      armChildTranDef = tranFactory.newArmTransactionDefinition(
         // only provide a name, no properties or ID
         armAppDef, "JExampleChildTx", null, null);
```

```
   armApp = tranFactory.newArmApplication(
      // only provide the definition and a group name (optional)
      armAppDef, "JExamples", null, null);
}

public void run() {
   armRegisterAndInit();

   ArmTransaction armParentTran = tranFactory.newArmTransaction(
      armApp, armParentTranDef);

   armParentTran.start();

   int childStatus = ArmConstants.STATUS_GOOD;
   if (! childTransaction(armParentTran.getCorrelator()))
      childStatus = ArmConstants.STATUS_FAILED;

   armParentTran.stop(childStatus);
}

/* In-process nested transaction. The return value signals
 * successful completion status.
 */
private boolean childTransaction(ArmCorrelator correlator) {
   ArmTransaction armChildTran = tranFactory.newArmTransaction(
      armApp, armChildTranDef);

   /* Instead of getting a correlator object from the parent level,
    * the correlator information might have been provided as collection
    * of bytes. In this case, a correlator would be constructed like:
    *
    * correlator = tranFactory.newArmCorrelator(correlatorByteArray);
    */

   armChildTran.start(correlator);

   /* If a remote (grand)child transaction exists, the current
    * transaction's correlator content should be extractes as byte array
    * and transferred to the child level like whown below. Note that
    * the messageToChild object and its methods are fictional,
    * representing some synchronous application-level message exchange.
    *
    * byte[] corrBytes = armChildTran.getCorrelator().getBytes();
    * messageToChild.appendToPayload(TAG_PAYLOAD_CORRELATOR, corrBytes);
    * messageToChild.send();
    * messageFromChild.receive();
    */

   armChildTran.stop(ArmConstants.STATUS_GOOD);

   return true;
}

/**
 * @param args
 */
public static void main(String[] args) {
   Arm40BasicExample example = new Arm40BasicExample();
   example.run();
```

```
   }

}
```

A.2 Detailed Timing and Threading Measurements

```
/* ------------------------------------------------------------------------ */
/* ---- ARM 4.1 Java API Example: using timing/threading capabilities  ---- */
/* ------------------------------------------------------------------------ */

package org.opengroup.arm40.examples;

import org.opengroup.arm40.transaction.ArmApplication;
import org.opengroup.arm40.transaction.ArmApplicationDefinition;
import org.opengroup.arm40.transaction.ArmConstants;
import org.opengroup.arm40.transaction.ArmCorrelator;
import org.opengroup.arm40.transaction.ArmTimestamp;
import org.opengroup.arm40.transaction.ArmTransaction;
import org.opengroup.arm40.transaction.ArmTransactionDefinition;
import org.opengroup.arm40.transaction.ArmTransactionFactory;

public class Arm40TimingThreadingExample {

   ArmTransactionFactory tranFactory;
   ArmApplicationDefinition armAppDef;
   ArmApplication armApp;
   ArmTransactionDefinition armParentTranDef;

   private ArmTransactionDefinition armChildTranDef;

   /* The run() method of this class will be executed in different thread than
    * the parent thread.
    */
   class ChildRunnable extends Thread {
      ArmTimestamp arrivalTimestamp;
      boolean state;
      private ArmCorrelator parentCorr;

      /* In order to take into account the time that elapses between
       * creation of the child thread and the actual start of the run()
       * method, the "arrival time" is measured when the new thread is
       * created and handed to the child instance.
       */
      ChildRunnable(ArmTimestamp initArrivalTS, ArmCorrelator initParentCorr) {
         arrivalTimestamp = initArrivalTS;
         parentCorr = initParentCorr;
      }

      void perform_rmi() { /* perform a blocking remote invocation here */}

      boolean getResult() {return state;}

      public void run() {
         ArmTransaction armChildTran = tranFactory.newArmTransaction(
            armApp, armChildTranDef);

         armChildTran.setPrestartTimeValue(arrivalTimestamp);

         armChildTran.start(parentCorr);
```

```
      armChildTran.bindThread();

      /* indicate a blocked status during a remote synchronous call */
      long blockHandle = armChildTran.blocked(); /* Spec: block() */
      perform_rmi();
      armChildTran.unblocked(blockHandle);  /* Spec: unblock() */

      /* not really necessary here because stop() implies unbindThread() */
      armChildTran.unbindThread();

      armChildTran.stop(ArmConstants.STATUS_GOOD);
   }
}

/* initialize factories, ARM definitions and an ARM application instance */
void armRegisterAndInit() {
   String tranFactoryName = System.getProperties().getProperty(
      ArmTransactionFactory.propertyKey,
      "org.opengroup.arm40.sdk.ArmTransactionFactoryImpl");

   try {
      tranFactory = (ArmTransactionFactory)
         Class.forName(tranFactoryName).newInstance();
   } catch (Exception e) {
      // Note: this example does not provide error handling at all
      System.exit(-1);
   }

   //
   armAppDef = tranFactory.newArmApplicationDefinition(
      // provide a name, no properties or ID
      "ARM Java Spec Timing/Threading Example", null, null);

   armParentTranDef = tranFactory.newArmTransactionDefinition(
      // only provide a name, no properties or ID
      armAppDef, "JExampleParentTx", null, null);

   armChildTranDef = tranFactory.newArmTransactionDefinition(
      // only provide a name, no properties or ID
      armAppDef, "JExampleChildTx", null, null);

   armApp = tranFactory.newArmApplication(
      // only provide the definition and a group name (optional)
      armAppDef, "JExamples", null, null);
}

public void run() {
   armRegisterAndInit();

   ArmTransaction armParentTran = tranFactory.newArmTransaction(
      armApp, armParentTranDef);

   armParentTran.start();

   int childStatus = ArmConstants.STATUS_GOOD;
   if (! dispatchChildTransactionToThread(armParentTran.getCorrelator()))
      childStatus = ArmConstants.STATUS_FAILED;

   armParentTran.stop(childStatus);
}
```

```
   /* For simplicity, this method cerates a single ChildRunnable
    * instance extending the Thread class and has it execute
    * on behalf of the child transaction.
    */
   private boolean dispatchChildTransactionToThread(
         ArmCorrelator parentCorrelator) {
      ArmTimestamp arrivalTimestamp = new ArmTimestamp();
      /* capure the current time */
      arrivalTimestamp.set();

      ChildRunnable childRunnable =
         new ChildRunnable(arrivalTimestamp, parentCorrelator);

      childRunnable.start();

      try {
         childRunnable.join();
      } catch (InterruptedException e) {
      }

      return childRunnable.getResult();
   }

   /**
    * @param args
    */
   public static void main(String[] args) {
      Arm40TimingThreadingExample example = new Arm40TimingThreadingExample();
      example.run();
   }

}
```

A.3 Messaging

```
/* ------------------------------------------------------------------------ */
/* ------- ARM 4.1 Java API Example: using messaging capabilities  -------- */
/* ------------------------------------------------------------------------ */

package org.opengroup.arm40.examples;

import org.opengroup.arm40.transaction.ArmApplication;
import org.opengroup.arm40.transaction.ArmApplicationDefinition;
import org.opengroup.arm40.transaction.ArmConstants;
import org.opengroup.arm40.transaction.ArmCorrelator;
import org.opengroup.arm40.transaction.ArmMessageEventGroup;
import org.opengroup.arm40.transaction.ArmMessageReceivedEvent;
import org.opengroup.arm40.transaction.ArmMessageSentEvent;
import org.opengroup.arm40.transaction.ArmTransaction;
import org.opengroup.arm40.transaction.ArmTransactionDefinition;
import org.opengroup.arm40.transaction.ArmTransactionFactory;

public class Arm40MessagingExample {

   ArmTransactionFactory tranFactory;
   ArmApplicationDefinition armAppDef;
   ArmApplication armApp;
   ArmTransactionDefinition armParentTranDef;
```

```
private ArmTransactionDefinition armChild1TranDef;
private ArmTransactionDefinition armChild2TranDef;

Thread child1;
Thread child2;

class PseudoMessage {
   /* assume some application payload data */
   public ArmCorrelator corr;

   /* wake up one erceiver of this object */
   synchronized void send() {
      notify();
   }

   /* wait for someone to send this object */
   synchronized boolean receive() {
      try {
         wait();
         return true;
      } catch (InterruptedException e) {
         return false;
      }
   }
}

PseudoMessage messageSentFromParent = new PseudoMessage();
PseudoMessage messageSentFromChild = new PseudoMessage();

/* initialize factories, ARM definitions and an ARM application instance */
void armRegisterAndInit() {
   String tranFactoryName = System.getProperties().getProperty(
      ArmTransactionFactory.propertyKey,
      "org.opengroup.arm40.sdk.ArmTransactionFactoryImpl");

   try {
      tranFactory = (ArmTransactionFactory)
         Class.forName(tranFactoryName).newInstance();
   } catch (Exception e) {
      // Note: this example does not provide error handling at all
      System.exit(-1);
   }

   //
   armAppDef = tranFactory.newArmApplicationDefinition(
      // provide a name, no properties or ID
      "ARM Java Spec Messaging Example", null, null);

   armParentTranDef = tranFactory.newArmTransactionDefinition(
      // only provide a name, no properties or ID
      armAppDef, "JExampleParentTx", null, null);

   armChild1TranDef = tranFactory.newArmTransactionDefinition(
      // only provide a name, no properties or ID
      armAppDef, "JExampleChildTx1", null, null);

   armChild2TranDef = tranFactory.newArmTransactionDefinition(
      // only provide a name, no properties or ID
      armAppDef, "JExampleChildTx2", null, null);
```

```
      armApp = tranFactory.newArmApplication(
         // only provide the definition and a group name (optional)
         armAppDef, "JExamples", null, null);
   }

   public void run() {
      armRegisterAndInit();

      ArmTransaction armParentTran = tranFactory.newArmTransaction(
         armApp, armParentTranDef);

      /* prepare a message sent event object and a containig group */
      ArmMessageSentEvent msgTxEv = tranFactory.newMessageSentEvent();
      msgTxEv.setMessageSentCount(1);
      msgTxEv.setDescription("parent message");

      ArmMessageEventGroup msgEvGroup = tranFactory.newMessageEventGroup();
      msgEvGroup.setEvent(0, msgTxEv);

      createChildren();

      armParentTran.start();

      messageSentFromParent.corr = armParentTran.getCorrelator();
      /* trigger the "receipt" of the message by child1 */
      messageSentFromParent.send();
      armParentTran.setMessageEventGroup(msgEvGroup);
      /* process the message sent event */
      armParentTran.update();

      waitForChildren();

      /* the stop time take at the parent level is meaningless for
       * the downstream event flow which ends at the second level child.
       */
      armParentTran.stop(ArmConstants.STATUS_GOOD);
   }

   /* waits for child1 and child2 to finish */
   private void waitForChildren() {
   }

   private void createChildren() {

      /* child1 blocks waiting for a message from parent then sends
       * a message to child2.
       */
      child1 = new Thread() {
         public void run() {
            ArmTransaction armChildTran = tranFactory.newArmTransaction(
               armApp, armChild1TranDef);

            /* prepare message event objects and a containig group */
            ArmMessageReceivedEvent msgRxEv =
tranFactory.newMessageReceivedEvent();
            msgRxEv.setCorrelatorReceived(messageSentFromParent.corr);
            msgRxEv.setDescription("parent message");

            ArmMessageSentEvent msgTxEv = tranFactory.newMessageSentEvent();
```

```
            msgTxEv.setMessageSentCount(1);
            msgTxEv.setDescription("child1 message");

            ArmMessageEventGroup msgEvGroup =
tranFactory.newMessageEventGroup();
            msgEvGroup.setEvent(0, msgRxEv);

            /* msgEvGroup will be processed by start() below */
            armChildTran.setMessageEventGroup(msgEvGroup);

            /* blocked waiting */
            messageSentFromParent.receive();
            armChildTran.start();

            /* re-use the group, it will be processed by stop() below */
            msgEvGroup.setEvent(0, msgTxEv);
            armChildTran.setMessageEventGroup(msgEvGroup);
            messageSentFromChild.corr = armChildTran.getCorrelator();
            messageSentFromChild.send();

            armChildTran.stop(ArmConstants.STATUS_GOOD);
         }
      };

      /* child1 blocks waiting for a message from child1 */
      child2 = new Thread() {
         public void run() {
            ArmTransaction armChildTran = tranFactory.newArmTransaction(
               armApp, armChild2TranDef);

            /* prepare message event objects and a containig group */
            ArmMessageReceivedEvent msgRxEv =
tranFactory.newMessageReceivedEvent();
            msgRxEv.setCorrelatorReceived(messageSentFromParent.corr);
            msgRxEv.setDescription("parent message");

            ArmMessageEventGroup msgEvGroup =
tranFactory.newMessageEventGroup();
            msgEvGroup.setEvent(0, msgRxEv);
            /* the receipt of a message by child2 marks the end of the
             * processing chain, which is reflected by the "End of Flow" flag.
             */
            msgEvGroup.setEndOfFlow(true);

            /* msgEvGroup will be processed by start() below */
            armChildTran.setMessageEventGroup(msgEvGroup);

            /* blocked waiting */
            messageSentFromChild.receive();
            armChildTran.start();

            armChildTran.stop(ArmConstants.STATUS_GOOD);
         }
      };
   }

   /**
    * @param args
    */
   public static void main(String[] args) {
```

```
      Arm40MessagingExample example = new Arm40MessagingExample();
      example.run();
   }

}
```

A.4 Instrumentation Control

```
/* ------------------------------------------------------------------------ */
/* ------- ARM 4.1 Java API Example: using instrumentation control -------- */
/* ------------------------------------------------------------------------ */

package org.opengroup.arm40.examples;

import org.opengroup.arm40.transaction.ArmApplication;
import org.opengroup.arm40.transaction.ArmApplicationControl;
import org.opengroup.arm40.transaction.ArmApplicationDefinition;
import org.opengroup.arm40.transaction.ArmConstants;
import org.opengroup.arm40.transaction.ArmCorrelator;
import org.opengroup.arm40.transaction.ArmTransaction;
import org.opengroup.arm40.transaction.ArmTransactionDefinition;
import org.opengroup.arm40.transaction.ArmTransactionFactory;
import org.opengroup.arm40.transaction.ArmTransactionDefinitionControl;

public class Arm40ControlExample {

   /* call childTransaction() N times */
   private static final int TOTAL_CHILD_CALLS = 0;
   /* check every N transactions */
   private static final int CHECK_CONTROL_INTERVAL=1000;

   ArmTransactionFactory tranFactory;
   ArmApplicationDefinition armAppDef;
   ArmApplication armApp;
   ArmTransactionDefinition armParentTranDef;

   private ArmTransactionDefinition armChildTranDef;

   /* Flags reflecting the instrumentation control state */
   boolean enableGlobal;
   boolean enableChildTran;

   /* result of the appliaction level control feedback are stored here */
   ArmApplicationControl armAppControl;

   /* initialize factories, ARM definitions and an ARM application instance */
   void armRegisterAndInit() {
      String tranFactoryName = System.getProperties().getProperty(
         ArmTransactionFactory.propertyKey,
         "org.opengroup.arm40.sdk.ArmTransactionFactoryImpl");

      try {
         tranFactory = (ArmTransactionFactory)
            Class.forName(tranFactoryName).newInstance();
      } catch (Exception e) {
         // Note: this example does not provide error handling at all
         System.exit(-1);
      }
```

```
      //
      armAppDef = tranFactory.newArmApplicationDefinition(
         // provide a name, no properties or ID
         "ARM Java Spec Instrumentation Control Example", null, null);

      armParentTranDef = tranFactory.newArmTransactionDefinition(
         // only provide a name, no properties or ID
         armAppDef, "JExampleParentTx", null, null);

      armChildTranDef = tranFactory.newArmTransactionDefinition(
         // only provide a name, no properties or ID
         armAppDef, "JExampleChildTx", null, null);

      armApp = tranFactory.newArmApplication(
         // only provide the definition and a group name (optional)
         armAppDef, "JExamples", null, null);

   }

   public void run() {
      armRegisterAndInit();

      /* Default: if no instrumentation control is provided, enable everything
*/
      enableGlobal = true;
      enableChildTran = true;

      /* Check for ARM instrumentation control support */
      armAppControl = armApp.getApplicationControl();
      /* null means no instrumentation control is provided */
      if (armAppControl != null) {
         /* For simplicity, this example interprets only on/off granularity */
         if (armAppControl.getCollectionDepth() ==
ArmConstants.COLLECTION_DEPTH_NONE) {
            enableGlobal = false;
            enableChildTran = false;
         }
      }

      ArmTransaction armParentTran = tranFactory.newArmTransaction(
         armApp, armParentTranDef);

      if (enableGlobal)
         armParentTran.start();

      int childStatus = ArmConstants.STATUS_GOOD;
      if (! childTransaction(armParentTran.getCorrelator()))
         childStatus = ArmConstants.STATUS_FAILED;

      if (enableGlobal)
         armParentTran.stop(childStatus);
   }

   /* In-process nested transaction. The return value signals
    * successful completion status.
    */
   private boolean childTransaction(ArmCorrelator correlator) {
      ArmTransaction armChildTran = tranFactory.newArmTransaction(
         armApp, armChildTranDef);
```

```
      for(int i=0; i<TOTAL_CHILD_CALLS; ++i) {
         /* is instrumenation control supported at all? */
         if ((i%CHECK_CONTROL_INTERVAL == 0) && (armAppControl != null) &&
               (armAppControl.isTransactionDefinitionControlUsed())) {
            ArmTransactionDefinitionControl tranDefinitionControl =
               armApp.getTransactionDefinitionControl(armChildTranDef.getID());

            /* override application control settings */
            if ((tranDefinitionControl != null) &&
               (tranDefinitionControl.getCollectionDepth() ==
                  ArmConstants.COLLECTION_DEPTH_NONE))
               enableChildTran = false;
            else
               enableChildTran = true;
         }

         if (enableChildTran)
            armChildTran.start(correlator);

         /* Child transaction work is performed here.*/

         if (enableChildTran)
            armChildTran.stop(ArmConstants.STATUS_GOOD);
      }

      return true;
   }

   /**
    * @param args
    */
   public static void main(String[] args) {
      Arm40ControlExample example = new Arm40ControlExample();
      example.run();
   }

}
```

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 (Java Virtual Machine), 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 are written in Java, the JVM would ensure that the same ordering conventions are followed and no order would need to be specified. However, correlators are meant to be passed between applications using any version of ARM (both C and Java) and running on any platform, including both big-endian and little-endian platforms. Because big-endian and little-endian platforms order bytes differently, the specification needs to explicitly state the required ordering, in order to make the correlators interchangeable.

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. For example, an application using ARM 4.0 Java Bindings 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 Java 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.

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 Limits on Interoperability between ARM Implementations

There is one limit on interoperability. In ARM 2.0 and 3.0, the maximum length of a correlator is 168 bytes. ARM 4.0 has changed the maximum to 512 bytes. If ARM 4.0 implementations restrict themselves to correlators of no more than 168 bytes, then the correlators are fully

interchangeable with any other version of ARM. If an ARM 4.0 implementation uses a correlator that is more than 168 bytes long, it can only be successfully interchanged with another ARM 4.0 implementation.

B.3 Correlator Formats

ARM specifies formatting constraints that all correlators must adhere to. These are described in the following section. In addition, different versions of ARM have defined three specific correlator formats. ARM 4.0 has not defined any formats, and, in general, has taken the approach of making correlators as opaque as possible.

B.4 ARM Correlator Format Constraints

These constraints apply to all formats.

Table 7: 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 <= 512. Lengths shorter than four bytes are not permitted because all correlators must have the four bytes defined in this table. Note that a correlator that is longer than 168 bytes could not be passed to and used by an application using ARM 2.0 or ARM 3.0, because the maximum size in those versions was 168 bytes. Some correlator formats impose shorter length restrictions. In particular, formats 1, 2, and 127 have a maximum of 168 bytes.
Byte 2	1 byte	Correlator format. The range 0:127 (unsigned) is reserved by the ARM specification. Six values have been assigned: 1 – Defined in ARM 2.0 2 – Defined in ARM 3.0 28 – Reserved for Hewlett-Packard 100 – Reserved for MyARM's implementation 103 – Reserved for IBM 122 – Reserved for tang-IT 127 – Defined in ARM 3.0 The range 128:255 (unsigned) is available for use by ARM implementers. Known used values include: 128 – Hewlett-Packard 203 – IBM 204 – IBM
Byte 3	1 byte	Flags All eight-bit flags are reserved by the ARM specification. Four flags are defined in positions 0:3 (the highest order bits), as in *abcd*0000, where *a, b, c,* and *d* are bit flags.

Position	Length	Contents
		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. There are no requirements for how these flags are handled, if at all. By convention, if the flags are turned on in a correlator, the setting is copied into the correlators for child transactions. However, local policy or the ARM implementation may override this convention. 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 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 ARM specification. c = 1 if the control flow from the parent was accomplished via an asynchronous mechanism such as a messaging protocol OR that the transaction flow is asynchronous in nature. See Section 4.2.1 for a more complete description. *d* = 1 indicates that ARM-reported program logic execution in the child recipient of this correlator reflects transaction work that, although initiated by this parent, is performed independently of the parent's ARM-reported transaction work in scope and purpose. See Section 4.2.2 for a more complete description. It logically follows that the child transaction is executing asynchronously to the parent transaction and therefore d=1 only if c=1. If c=0, d is ignored.

Index