Politecnico di Milano
A.A. 2015-2016

Software Engineering 2

Code Inspection

Matteo Bulloni (852676), Marco Cannici (852527)

5 January 2016

Contents

1 Assigned class and methods

The class assigned to us is called "**CurrentTransaction**" (namespace: `com.sun.jts.CosTransactions.CurrentTransaction`) and is located in the following path, relative to the root of GlassFish project: `appserver/transaction/jts/src/main/java/com/sun/jts/CosTransactions/CurrentTransaction.java`

The following are the methods of the "**CurrentTransaction**" class assigned to us:

- Name: ***endAborted****(boolean [ ] aborted , boolean endAssociation)*
  Start Line: 374
- Name: ***sendingReply****(int id , PropagationContextHolder holder)*
  Start Line: 1035

2 Functional Role

2.1 JTS Transaction Service

The class **CurrentTransaction** assigned to us is part of the Java™ Transaction Service (JTS) implementation by Oracle.

The "*Java™ Transaction Service (JTS) Specification*"[2] says:

> JTS specifies the implementation of a transaction manager which supports the JTA (Java Transaction API) specification at the high-level and implements the Java mapping of the OMG Object Transaction Service (OTS) 1.1 Specification at the low-level.

The Object Transaction Service is a paradigm that allows distributed access to resorces and computation (remote method calls).[1]

2.2 CurrentTransaction class

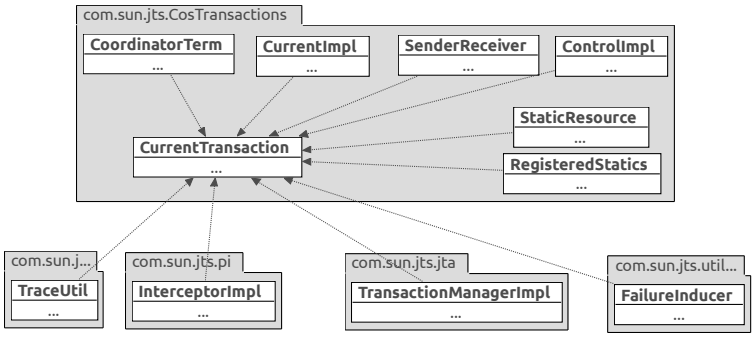
The "CurrentTransaction" class is a static class that does not implement any interface and is used to keep track of the associations between transactions and threads.

The following is the JavaDoc of the class:

```
81 /**This class manages association of transactions with threads in a process,
82  * and associated state/operations.
83  *
84  * @version 0.01
85  *
86  * @author Simon Holdsworth, IBM Corporation
87  *
88  * @see
89 */
```

For each thread the class keeps track of the transactions with which it is associated to, the list of suspended transactions (which are transactions that have been suspended because a new request has been received while they were running) and the list of RegisteredStatics objects that will be informed of any changes in the associations of the thread with the transactions. The class exposes methods to modify the current association of the thread and the list of suspended transactions and to retrieve the list of transactions associated to the current thread. It also exposes methods to notify the Control object that a reply or a request has been (or is about to be) either received or sended. The Control object associated to each transaction allows access to a Terminator object (which provides methods for commit or rollback) and a Coordinator object (which involves Resource objects in a transaction when they are registered[1]).

The following is a class diagram showing the main classes with which CurrentTransaction class interacts with:



2.3 endAborted method

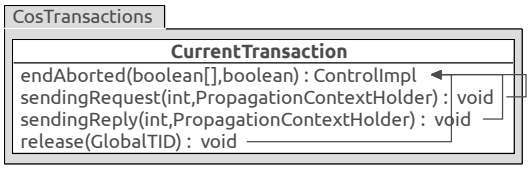
This is a private method of the class "CurrentTransaction", and it is used to ensure that the Control object associated with the current thread does not represent a transaction that has already been aborted. To do so, if the Control object does represent an aborted transaction, the method terminates the current association and replaces it with an active (not aborted) one.

The following are the JavaDoc and the declaration of the method:

```
353     /**Ensures that an association with an aborted transaction is dealt with
        ↪ cleanly.
354      *
355      *
356      * TN - do not dissociate thread even if it's aborted!!
357      *
358      * If the current Control object represents a transaction that has been
359      * aborted, this method replaces the association by one with the first
360      * ancestor that has not been aborted, if any, or no association, and the
361      * method returns true as the output parameter. Otherwise the method
        ↪ returns
362      * false as the output parameter.
363      * <p>
364      * If there is a current Control object in either case it is returned,
365      * otherwise null is returned.
366      *
367      * @param aborted  A 1-element array which will hold the aborted
        ↪ indicator.
368      *
369      * @return  The current Control object.
370      *
371      * @see
372      */
373     private static ControlImpl
374         endAborted( boolean[/*1*/] aborted, boolean endAssociation) {
```

The method checks if the transaction associated with the current thread has already been aborted (communicating it to the caller through the boolean output parameter "**aborted**", which is an improperly used single element array instead of a proper Boolean object) by checking the transaction's status. If that's the case, and if the method is called with "**endAssociation**" argument set to true, *endAborted()* replaces the Control object associated with the current thread with its first ancestor that has not been aborted, by calling *popAborted()*

's Control method to resume it. Eventually, *endAborted()* method also deals with informing all the registered StaticResource objects that the old thread association has been terminated and a new one has been established. However, if *endAborted()* is called with "**endAssociation**" argument set to false instead, it means that it's being used just to check that the thread is associated to an active transaction, which is not its specific purpose. This is kind of an improper use of the method: it would have probably been better to create a method to specifically accomplish this task.



This method is used by the public and friendly methods *release()*, *sendingReply()* and *sendingRequest()* of CurrentTransaction class. To show how the method is actually used, we report the code snippets of the methods listed above showing where it is called:

```
776         // Ensure that the current Control object is valid.  Return
            ↪ immediately if
777         // not.
778
779         boolean[] outBoolean = new boolean[1];
780         ControlImpl current = endAborted(outBoolean, false);
781         if( outBoolean[0] ) {
782             TRANSACTION_ROLLEDBACK exc = new TRANSACTION_ROLLEDBACK(0,
                ↪ CompletionStatus.COMPLETED_NO);
783             throw exc;
784         }
```

Listing 1: sendingRequest() calls endAboted()

```
1059        // Ensure that the current Control object is valid.  Return
            ↪ immediately if not.
1060
1061        boolean[] outBoolean = new boolean[1];
1062        ControlImpl current = endAborted(outBoolean, true);  // end
            ↪ association
1063        if( outBoolean[0] ) {
1064            importedTransactions.remove(Thread.currentThread());
1065            TRANSACTION_ROLLEDBACK exc = new TRANSACTION_ROLLEDBACK(0,
                ↪ CompletionStatus.COMPLETED_YES);
1066            throw exc;
1067        }
```

Listing 2: sendingReply() calls endAborted()

```
1256        // Ensure that the current Control object is valid.
1257        boolean[] outBoolean = new boolean[1];
1258        ControlImpl control = endAborted(outBoolean, true);  // end
            ↪ association
1259        if (outBoolean[0]) {
```

```
1260            importedTransactions.remove(Thread.currentThread());
1261        return; // thread is not associated with tx, simply return
1262          }
```

Listing 3: release() calls endAborted()

As we can see in Listing 1, *endAborted()* method is sometimes invoked just to check that the current thread is associated with an active transaction; the check is performed by looking at the output parameter **outBoolean**. It can also be seen that, whenever it is expected that the current thread is associated with an active transaction but it turns out that it's not, the methods that invoked *endAborted()* return immediately, throwing an exception to communicate the unexpected behaviour.

2.4 sendingReply method

This is a public method of the "CurrentTransaction" class and it is called to inform the Coordinator of the current transaction that an imminent reply is about to be performed and so the association between the transaction and the current thread should be ended.

The following are the JavaDoc and the declaration of the method:

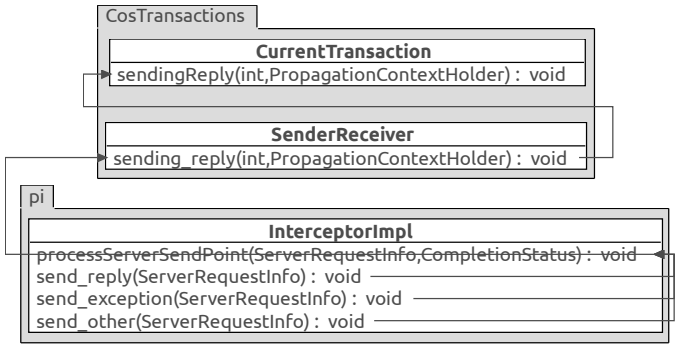
```
1021    /**Informs the object's Coordinator that a reply is being sent to the
        ↪ client.
1022     *
1023     * @param id      The request identifier.
1024     * @param holder  The context to be returned on the reply.
1025     *
1026     * @exception INVALID_TRANSACTION  The current transaction has
        ↪ outstanding work
1027     *   on this reply, and has been marked rollback-only, or the reply is
        ↪ returning
1028     *   when a different transaction is active from the one active when the
        ↪ request
1029     *   was imported.
1030     * @exception TRANSACTION_ROLLEDBACK  The current transaction has already
        ↪  been
1031     *   rolled back.
1032     *
1033     * @see
1034     */
1035    static void sendingReply( int id,
1036                                          PropagationContextHolder holder )
1037        throws INVALID_TRANSACTION, TRANSACTION_ROLLEDBACK {
```

The method is responsible to check that the current transaction is actually still active and there are no pending computations that must be terminated. To accomplish the first task the "**endAborted()**" method (see section 2.3 for a more detailed explanation) is called to check if the transaction has already been aborted, and if so a *TRANSACTION_ROLLBACK* exception is raised communicating that the transaction is already completed (*CompletionStatus.COMPLETED_YES*) and the Coordinator is set to rollback only mode by calling *rollback_only()* method. For what concerns the second task, the method requests the status of the Coordinator by calling his *replyAction()* method:

- If there are still subtransactions that haven't been completed yet (the value *CoordinatorImpl.activeChildren* has been returned), an *INVALID_ TRANSACTION* exception is raised communicating the error code "*MinorCode.UnfinishedSubtransactions*"

- If the transaction is still associated to a thread different from the current one or there are outgoing requests of the Coordinator that have not been completed yet, an *INVALID_ TRANSACTION* exception is raised communicating the error code "*MinorCode.DeferredActivities*"

Finally, the method deals with terminating the association with the transaction keeping consistent the list of transactions associated with the current thread, and resuming the last transaction that had been suspended by calling "*endCurrent()*" method.



The *sendingReply()* method is called whenever the methods *send_reply()*, *send_exception()* and *send_other()* of the **InterceptorImpl** class are invoked passing through *processServerSendPoint()* and *sending_reply()* calls, as shown in the class diagram above.

A detailed explanation of what Interceptors are and when *send_reply()*, *send_exception()* and *send_other()* are called is provided in "*Transaction Service Specification*"[1] and "CORBA Request Portable Interceptors: A Performance Analysis"[3] documents. We report below the most significant parts:

> Portable Request Interceptors (PIs) are a mechanism allowing to modify the ORB or the application behaviour upon the event of sending or receiving a message (e.g. a request, a reply or an exception) without impacting either on the ORB code or on the application one.
>
> The Transaction Service and the ORB must cooperate to realize certain Transaction Service function. This cooperation is realized on the **client invocation path** and through the transaction interceptor.

Request Interceptors are classified in client request interceptors and server request interceptors. The former are installed in client-side ORBs and can intercept outgoing requests and contexts as well as incoming replies and exceptions. Conversely, the latter are installed in server-side ORBs and can intercept incoming requests and contexts as well as outgoing replies and exceptions.
Server request interceptors are activated either upon receiving a request (by implementing the receive_request(), receive_poll() or receive_request_service_contexts()) or upon the sending of a reply or of an exception (by implementing the **send_reply()**, **send_exception()** or **send_other()** methods).

3 Code Inspection

3.1 Class Analysis

1. **All class names, interface names, method names, class variables, method variables, and constants used should have meaningful names and do what the name suggests.**

   - method names:
     - line 1302: "endAll()" method is not implemented
     - line 1358: "shutdown()" method is not implemented
     - line 1371: "dump()" method is not implemented
   - method variables:
     - line 750: the argument "id" of the method "sendingRequest" is never being used inside the function, so it could be removed

```
750     static void sendingRequest( int id,
751                                                 PropagationContextHolder
                                                    ↪ holder )
752         throws TRANSACTION_ROLLEDBACK, TRANSACTION_REQUIRED {
```

6. **Class variables, also called attributes, are mixed case, but might begin with an underscore ('_') followed by a lowercase first letter. All the remaining words in the variable name have their first letter capitalized. Examples: _windowHeight, timeSeriesData.**

   - line 111: the variable "m_tid" doesn't respect the naming convention because the underscore should only appear at the beginning of the name

```
111     private static ThreadLocal m_tid=new ThreadLocal();
```

23. **Check that the javadoc is complete (i.e., it covers all classes and files part of the set of classes assigned to you)**.

    For the following public and friendly methods and class variables is not provided a javadoc documentation:

    - line 346: "isTxAssociated()" public method

```
343     // COMMENT (Ram J) 12/18/2000
344     // This is being accessed from OTS interceptors package to
345     // check to see if there is a current transaction or not.
346     public static boolean isTxAssociated() {
```

- line 102 “statsOn()” friendly method

```
102     static boolean statsOn=false;
```

- line 119 “_logger”: firendly class variable

```
119   static Logger _logger = LogDomains.getLogger(CurrentTransaction.
      ↪ class, LogDomains.TRANSACTION_LOGGER);
```

The following are methods for which it is reported a javadoc documentation but the meaning of some arguments or thrown exception is not clarified:

- line 374: the meaning of the argument “endAssociation” is not provided in the documentation

```
353     /**Ensures that an association with an aborted transaction is
        ↪ dealt with cleanly.
354      *
355      *
356      * TN - do not dissociate thread even if it's aborted!!
357      *
358      * If the current Control object represents a transaction that
         ↪  has been
359      * aborted, this method replaces the association by one with
         ↪ the first
360      * ancestor that has not been aborted, if any, or no
         ↪ association, and the
361      * method returns true as the output parameter. Otherwise the
         ↪ method returns
362      * false as the output parameter.
363      * <p>
364      * If there is a current Control object in either case it is
         ↪ returned,
365      * otherwise null is returned.
366      *
367      * @param aborted  A 1-element array which will hold the
         ↪ aborted indicator.
368      *
369      * @return  The current Control object.
370      *
371      * @see
372      */
373     private static ControlImpl
374         endAborted( boolean[/*1*/] aborted, boolean endAssociation
            ↪ ) {
```

- line 493: It is not specified when the method “getCurrent” could raise the exception TRANSACTION_ROLLEDBACK

```
480     /**Returns the current Control object.
481      * <p>
482      * That is, the Control object that corresponds to the thread
```

```
483      * under which the operation was invoked. If there is no such
         ↪ association the
484      * null value is returned.
485      *
486      * @param
487      *
488      * @return  The current Control object.
489      *
490      *
491      * @see
492      */
493     public static ControlImpl getCurrent()
494         throws TRANSACTION_ROLLEDBACK {
```

- line 1199: the meaning of the argument "timeout" is not provided in the documentation

```
1192     /**
1193      * Recreates a transaction based on the information contained
         ↪ in the
1194      * transaction id (tid) and associates the current thread of
         ↪ control with
1195      * the recreated transaction.
1196      *
1197      * @param tid  the transaction id.
1198      */
1199     public static void recreate(GlobalTID tid, int timeout) {
```

25. **The class or interface declarations shall be in the following order: [...]**

    The following class variables should be declared before the private ones because they are friendly (and moreover, as described in the point **28.**, these variables could be declared private because they are used only inside this class)

- line 102: "statsOn" friendly variable

```
101     //store the suspended and associated transactions support only
        ↪  if stats are required
102     static boolean statsOn=false;
```

- line 119: "_logger" friendly variable

```
116   /*
117     Logger to log transaction messages
118   */
119   static Logger _logger = LogDomains.getLogger(CurrentTransaction.
      ↪ class, LogDomains.TRANSACTION_LOGGER);
```

28. **Check that variables and class members are of the correct type. Check that they have the right visibility (public/private/protected).**

The following class variables could be declared private because they are only used inside the current class:

- line 102: “statsOn” friendly variable

```
102     static boolean statsOn=false;
```

- line 119: “_logger” friendly variable

```
119   static Logger _logger = LogDomains.getLogger(CurrentTransaction.
      ↪ class, LogDomains.TRANSACTION_LOGGER);
```

3.2 Method analysis: “endAborted”

```
353     /**Ensures that an association with an aborted transaction is dealt with
        ↪ cleanly.
354      *
355      *
356      * TN - do not dissociate thread even if it's aborted!!
357      *
358      * If the current Control object represents a transaction that has been
359      * aborted, this method replaces the association by one with the first
360      * ancestor that has not been aborted, if any, or no association, and the
361      * method returns true as the output parameter. Otherwise the method
         ↪ returns
362      * false as the output parameter.
363      * <p>
364      * If there is a current Control object in either case it is returned,
365      * otherwise null is returned.
366      *
367      * @param aborted  A 1-element array which will hold the aborted
         ↪ indicator.
368      *
369      * @return  The current Control object.
370      *
371      * @see
372      */
373     private static ControlImpl
374         endAborted( boolean[/*1*/] aborted, boolean endAssociation) {
375
376         // Get the current thread identifier, and the corresponding Control
            ↪ object
377         // if there is one.
378
379         boolean completed = true;
380         aborted[0] = false;
381
382         ControlImpl result = (ControlImpl)m_tid.get();
383
384         // If there is a current Control object, and it represents a
            ↪ transaction that
385         // has been aborted, then we need to end its association with the
            ↪ current
386         // thread of control.
387
388         if( result != null )
389             try {
390                 completed = (result.getTranState() != Status.StatusActive);
391             } catch( Throwable exc ) {
392     _logger.log(Level.FINE,"", exc);
393             }
394
395         if( result != null && completed ) {
396             if (endAssociation) {
397           synchronized(CurrentTransaction.class){
398       if(statsOn){
399                 Thread thread = Thread.currentThread();
400                       threadContexts.remove(thread);
401       }
402       m_tid.set(null);
403
404                   // XA support: If there was a current IControl, inform all
                      ↪ registered
405                   // StaticResource objects of the end of the thread
                      ↪ association.
406                   // Allow any exception to percolate to the caller.
407
408                   if( statics != null )
409                         statics.distributeEnd(result,false);
```

```
410
411                 // Discard all stacked controls that represent aborted or
                    ↪ unrecognised
412                 // transactions.
413
414                 result = result.popAborted();
415
416                 // If there is a valid ancestor, make it the current one.
417
418                 if( result != null ) {
419       m_tid.set(result);
420       if(statsOn){
421               Thread thread = Thread.currentThread();
422                         threadContexts.put(thread,result);
423                         suspended.removeElement(result);
424       }
425                 }
426
427                 // XA support: If there is a stacked context, inform all
                    ↪ registered
428                 // StaticResource objects of the new thread association.
429                 // Allow any exception to percolate to the caller.
430
431                 if( statics != null )
432                       statics.distributeStart(result,false);
433   }
434           }
435           aborted[0] = true;
436       }
437
438   if(_logger.isLoggable(Level.FINEST))
439   {
440     Thread thread = Thread.currentThread();
441     _logger.logp(Level.FINEST,"CurrentTransaction","endAborted()",
442         "threadContexts.get(thread) returned " +
443         result + " for current thread " + thread);
444   }
445
446       return result;
447   }
```

8. **Three or four spaces are used for indentation and done so consistently**
   Blocks of four spaces are used for indentation along the method (even if multiple times in the form of tab characters instead of spaces (see point **9.** below)), but many times the indentation rules are not applied correctly:

   - Line **392** not correctly indented
   - Content of if() block fromline **396** to **434** not correctly indented
   - Content of synchronized() block from line **397** to **433** and its closing bracket at line **433** not correctly indented
   - Content of if() block from line **398** to **401** not correctly indented
   - Content of if() block at line **408** not correctly indented
   - Content of if() block from line **418** to **425** not correctly indented
   - Content of if() block from line **420** to **424** not correctly indented
   - Content of if() block at line **431** not correctly indented

- Lines **442**, **443** not correctly indented

9. **No tabs are used to indent**

   Starting from line **398** until line **444**, lines **435**-6-7 excluded, each line that is not a blank line is indented using tabs instead of spaces.

10. **Consistent bracing style is used, either the preferred "Allman" style (first brace goes underneath the opening block) or the "Kernighan and Ritchie" style (first brace is on the same line of the instruction that opens the new block)**

    The author has used the "Kernighan and Ritchie" bracing style along all the method, except for the if() block from line **438** to line **444**, where he used the "Allman" style. This lack of consistency should be avoided.

11. **All if, while, do-while, try-catch, and for statements that have only one statement to execute are surrounded by curly braces**

    - if() block from line **388** to **393** not surrounded by curly braces
    - if() block from line **408** to **409** not surrounded by curly braces
    - if() block from line **431** to **432** not surrounded by curly braces

13. **Where practical, line length does not exceed 80 characters**
    All the lines of code of the method do not exceed 80 characters; however, some lines of either Javadoc or comments do:

    - Javadoc: line **353**
    - Comments: lines **376**, **384**, **385**, **404**, **405**, **411**, **427**

    The peak is at line **411**, which is 90 characters long.

15. **Line break occurs after a comma or an operator.**

    The method declaration itself, at line **373**, violates this rule: the line break there is in fact placed between the return type and the name of the method, and thus not after either a comma or an operator.

17. **A new statement is aligned with the beginning of the expression at the same level as the previous line**

    As already mentioned in point **8.** together with other indentation errors, there are some lines of subsequent instructions that should be aligned since they are at the same level, but are not:

    - Lines **399** and **400**
    - Lines **421**, **422**, **423**

- Lines **441**, **442**, **443**: lines **442** and **443** should be aligned with the open bracket at line **441**

40. **Check that all objects (including Strings) are compared with "equals" and not with "=="**

    The comparison between the two objects at line **390** doesn't follow this criterion, since it's done with operator "!=" instead of "!(equals(obj1,obj2))".

42. **Check that error messages are comprehensive and provide guidance as to how to correct the problem**

    The error message wrapped in the "try...catch" block that starts at line **389** doesn't explain anything at all about the problem related with that exception.

50. **Check throw--catch expressions, and check that the error condition is actually legitimate**

    The "try...catch" block that starts at line **389** doesn't seem to be useful, since the "try" block just includes an assignment of the result of a comparison between to objects to a boolean variable: since one of the two objects compared is a static final attribute and the other is the returned variable of a getter method that doesn't throw any exception, and always seems to just return an object correctly, it can't be explained which exception was meant to be caught here and why. This "try...catch" block thus appears to be meaningless, since no exception is expected to be thrown by those variable assignment and method call operations. Moreover, the lack of a specific declaration of the type of exception that was expected to be caught - the exception is just declared of "Throwable" type, which should be avoided - fails to provide any possible insight on this issue.

53. **Check that the appropriate action are taken for each catch block**

    Referring to the explanations provided in point 10. of this list, it's actually impossible to understand if a proper action is taken in the "catch" block at line **391**; however, since nothing is actually being done in that "catch" block except for logging the exception without any explanation, it is quite likely that it's not an appropriate way to handle the exception that was being expected there.

3.3 Method analysis: “sendingReply”

```
1021     /**Informs the object's Coordinator that a reply is being sent to the
         ↪ client.
1022      *
1023      * @param id      The request identifier.
1024      * @param holder  The context to be returned on the reply.
1025      *
1026      * @exception INVALID_TRANSACTION  The current transaction has
         ↪ outstanding work
1027      *   on this reply, and has been marked rollback-only, or the reply is
         ↪ returning
1028      *   when a different transaction is active from the one active when the
         ↪ request
1029      *   was imported.
1030      * @exception TRANSACTION_ROLLEDBACK  The current transaction has already
         ↪  been
1031      *   rolled back.
1032      *
1033      * @see
1034      */
1035     static void sendingReply( int id,
1036                                             PropagationContextHolder holder )
1037         throws INVALID_TRANSACTION, TRANSACTION_ROLLEDBACK {
1038
1039         // Zero out context information.
1040         // Ensure that the cached reference to the ORB is set up, and that
            ↪ the Any
1041         // value in the context is initialised.
1042         //$ The following is necessary for the context to be marshallable.
            ↪ It is a
1043         //$ waste of time when there is no transaction, in which case we
            ↪ should be
1044         //$ throwing the TRANSACTION_REQUIRED exception (?).
1045
1046         if( emptyContext.implementation_specific_data == null ) {
1047             ORB orb = Configuration.getORB();
1048             emptyContext.implementation_specific_data = orb.create_any();
1049             emptyContext.implementation_specific_data.insert_boolean(false);
1050         }
1051
1052         // COMMENT(Ram J) There is no need to send an empty context, if a tx
1053         // is not available. The PI based OTS hooks will not send a tx
            ↪ context
1054         // in the reply.
1055         /*
1056         holder.value = emptyContext;
1057         */
1058
1059         // Ensure that the current Control object is valid.  Return
            ↪ immediately if not.
1060
1061         boolean[] outBoolean = new boolean[1];
1062         ControlImpl current = endAborted(outBoolean, true);  // end
            ↪ association
1063         if( outBoolean[0] ) {
1064             importedTransactions.remove(Thread.currentThread());
1065             TRANSACTION_ROLLEDBACK exc = new TRANSACTION_ROLLEDBACK(0,
                ↪ CompletionStatus.COMPLETED_YES);
1066             throw exc;
1067         }
1068
1069         // Get the global identifier of the transaction that was imported
            ↪ into this
1070         // thread.  If there is none, that is an error.
1071
1072         Thread thread = Thread.currentThread();
```

```
1073            GlobalTID importedTID = (GlobalTID)importedTransactions.remove(thread
                ↪ );
1074
1075            // If there is no import information, and no current transaction,
                ↪ then return
1076            // the empty context.
1077
1078            if( importedTID == null && current == null ) {
1079                return;
1080            }
1081
1082            // Check that the current transaction matches the one that was
                ↪ imported.
1083
1084            StatusHolder outStatus = new StatusHolder();
1085            try {
1086                if( importedTID == null ||
1087                    current == null ||
1088                    !importedTID.isSameTID(current.getGlobalTID(outStatus)) ||
1089                    outStatus.value != Status.StatusActive ) {
1090                    INVALID_TRANSACTION exc = new INVALID_TRANSACTION(MinorCode.
                        ↪ WrongContextOnReply,CompletionStatus.COMPLETED_YES);
1091                    throw exc;
1092                }
1093            } catch( SystemException ex ) {
1094                _logger.log(Level.FINE,"", ex);
1095                INVALID_TRANSACTION exc = new INVALID_TRANSACTION(MinorCode.
                    ↪ WrongContextOnReply,CompletionStatus.COMPLETED_YES);
1096                throw exc;
1097            }
1098
1099            //$Get the Coordinator reference.
1100
1101            CoordinatorImpl coord = null;
1102            Coordinator coordRef = null;
1103            try {
1104                if (Configuration.isLocalFactory()) {
1105                    coord = (CoordinatorImpl) current.get_localCoordinator();
1106                } else {
1107                    coordRef = current.get_coordinator();
1108                    coord = CoordinatorImpl.servant(coordRef);
1109                }
1110
1111                //    _logger.log(Level.FINE,"Servant = "+coord);
1112
1113                // Check the Coordinator before sending the reply.
1114                // We must do this before ending the thread association to allow
                    ↪ the
1115                // Coordinator to take advantage of registration on reply if
                    ↪ available.
1116                // Note that if the Coordinator returns forgetMe, the global
                    ↪ identifier
1117                // will have been destroyed at this point.
1118
1119                CoordinatorImpl forgetParent = null;
1120                int[] outInt = new int[1];
1121                //StatusHolder outStatus = new StatusHolder();
1122                try {
1123                    forgetParent = coord.replyAction(outInt);
1124                } catch( Throwable exc ) {
1125                    _logger.log(Level.FINE,"", exc);
1126                }
1127
1128                int replyAction = outInt[0];
1129                if( replyAction == CoordinatorImpl.activeChildren ) {
1130                    try {
1131                        coord.rollback_only();
1132                    } catch( Throwable ex ) {
```

```
1133                    _logger.log(Level.FINE,"", ex);
1134                }
1135
1136                INVALID_TRANSACTION exc = new INVALID_TRANSACTION(MinorCode.
                    ↪ UnfinishedSubtransactions,
1137                                                                  CompletionStatus
                                                                      ↪ .
                                                                      ↪ COMPLETED_YES
                                                                      ↪ );
1138                throw exc;
1139            }
1140
1141            // End the current thread association.
1142
1143            endCurrent(false);
1144
1145            // If the transaction needs to be cleaned up, do so now.
1146            // We ignore any exception the end_current may have raised in
                ↪ this case.
1147            // The Control object is destroyed before the Coordinator so that
                ↪  it is not
1148            // in the suspended set when the Coordinator is rolled back.
1149
1150            if( replyAction == CoordinatorImpl.forgetMe ) {
1151                current.destroy();
1152                coord.cleanUpEmpty(forgetParent);
1153            }
1154
1155            // Otherwise, we have to check this reply.
1156
1157            else {
1158                if( current.isAssociated() ||
1159                        current.isOutgoing() ) {
1160                    try {
1161                        coord.rollback_only();
1162                    } catch( Throwable exc ) {
1163                        _logger.log(Level.FINE,"", exc);
1164                    }
1165
1166                    INVALID_TRANSACTION exc = new INVALID_TRANSACTION(
                        ↪ MinorCode.DeferredActivities,
1167                                                                      CompletionStatus
                                                                          ↪ .
                                                                          ↪ COMPLETED_YES
                                                                          ↪ );
1168                    throw exc;
1169                }
1170
1171                current.destroy();
1172            }
1173
1174        } catch( INVALID_TRANSACTION exc ) {
1175            throw exc;
1176        } catch( Unavailable exc ) {
1177            _logger.log(Level.FINE,"", exc);
1178            // Ignore
1179        } catch( SystemException exc ) {
1180            _logger.log(Level.FINE,"", exc);
1181            // Ignore
1182        }
1183
1184        // Create a context with the necessary information.
1185        // All we propagate back is the transaction id and implementation
            ↪ specific data.
1186
1187        holder.value = new PropagationContext(0,new TransIdentity(null,null,
            ↪ importedTID.realTID),
```

```
1188                                                    new TransIdentity[0],
                                                        ↪ emptyContext.
                                                        ↪ implementation_specific_data
                                                        ↪ );
1189
1190    }
```

5. **Method names should be verbs, with the first letter of each addition word capitalized.**

   - line **1048**: the called method "create_any()" should be renamed in "createAny()"
   - line **1049**: the called method "insert_boolean()" should be renamed in "insertBoolean()"
   - line **1105**: the called method "get_localCoordinator()" should be renamed in "getLocalCoordinator()"
   - line **1107**: the called method "get_coordinator()" should be renamed in "getCoordinator()"
   - line **1108**: the called method "servant()" should be renamed in "getServant()"
   - line **1131** and **1161**: the called method "rollback_only()" should be renamed

8. **Three or four spaces are used for indentation and done so consistently**

   Four spaces are used for indentation but in the following cases the indentation rules are not applied correctly:

   - line **1036** not correclty indented (1 more space)
   - line **1137** not correclty indented (2 more spaces)
   - line **1167** not correctly indented (2 more spaces)
   - line **1188** not correctly indented (2 more spaces)

13. **Where practical, line length does not exceed 80 characters.**

    - Some lines of the javadoc documentation of this method exceed 80 characters
    - Some lines of the following comment blocks exceed 80 characters length:
      - block from line **1040** to **1044**
      - comment at line **1059**
      - comment at line **1069**
      - comment at line **1075**

- block from line **1114** to **1116**
- block from line **1146** to **1147**
- comment at line **1185**

14. **When line length must exceed 80 characters, it does NOT exceed 120 characters.**

    - line **1090**: line length is 129 characters
    - line **1095**: line length is 125 characters

17. **A new statement is aligned with the beginning of the expression at the same level as the previous line.**

    - line **1036**: the argument "holder" should be indented at the same level of the argument "id"
    - line **1137**: the argument "CompletionStatus.COMPLETED_YES" should be indented at the same level of the previous argument
    - line **1159**: "current.isOutgoing()" should be indented at the same same level of "current.isAssociated()"
    - line **1167**: the argument "CompletionStatus.COMPLETED_YES" should be indented at the same level of the previous argument

19. **Commented out code contains a reason for being commented out and a date it can be removed from the source file if determined it is no longer needed.**

    - line **1056**: It is not specified a date after which the commented code can be deleted
    - line **1111**: It is not specified neither the reason nor the date
    - line **1121**: It is not specified neither the reason nor the date

29. **Check that variables are declared in the proper scope**

    - line **1102**: "coordRef" declaration should be moved inside the else block at line 1106 because the variable is used only there.

33. **Declarations appear at the beginning of blocks (A block is any code surrounded by curly braces "{" and "}"). The exception is a variable can be declared in a 'for' loop.**

    - line **1061**, **1062**, **1072**, **1073**, **1084**, **1101**, **1119**, **1120**: declarations should be moved at the beginning of the function (line **1038**)
    - line **1136**: the declaration of the exception should be moved at the beginning of the if() block at line **1130**. The exception could be also immediately thrown instead of assigning it to a temporary variable.

- line **1166**: the declaration of the exception should be moved at the beginning of the if() block at line **1160**. The exception could be also immediately thrown instead of assigning it to a temporary variable.

40. **Check that all objects (including Strings) are compared with "equals" and not with "=="**

    - line **1089**: "!=" is used instead of !equals()

42. **Check that error messages are comprehensive and provide guidance as to how to correct the problem**

    At lines **1094**, **1125**, **1133**, **1163**, **1177**and **1180** it is not provided an explanation for the logged exception

44. **Check that the implementation avoids "brutish programming"**

    - line **1065**: The constructor of TRANSACTION_ROLLEDBACK should be called using the constant "Undefined" declared in MinorCode class instead of "0"

51. **Check that the code is free of any implicit type conversions**

    - line **1062**: The called function "endAborted()" uses a one-element array to pass the boolean argument by reference. It should be better to use the object type Boolean in order to avoid indexes from going out-of-bounds
    - line **1123**: The called function "replyAction()" uses a one-element array to pass the integer argument by reference. It should be better to use the object type Integer in order to avoid indexes from going out-of-bounds

52. **Check that the relevant exceptions are caught**

    In some cases instead of catching a specific exception a more general one is caught:

    - line **1124**: it should be catched a "SystemException" instead of "Throwable"
    - line **1131** and **1161**: it should be catched an "Inactive" exception instead of "Throwable"

3.4 Other problems found

1. As already highlighted in section 3.2 "endAborted method", the function ***endAborted****(boolean [/*1*/] aborted , boolean endAssociation)* should be using an object of type "Boolean" instead of a boolean array of a single element to be passed the "aborted" parameter and allowed to modify it. Improperly using an array is dangerous, and differently from the correct Boolean object it might lead to issues related with array indexing mistakes.

2. As already pointed out in section 3.2 "endAborted method", this method is actually performing two very different functions depending on the value of "endAssociation" parameter: if it's set to "true" when *endAborted* is called, it means that the method has to actually perform the complex duty it was written for (described in detail in section 3.2), while if it's set to "false" it means that it's being used just as a very simple transaction's status checker. Since the two functions performed are very different, it should be implemented and used a new method explicitly designed to just perform the transaction's status check, while *endAborted* should be rewritten without the "endAssociation" parameter, meaning that it would just perform its specific duty, without the potential worry of causing any mistake because of a wrong parameter value.

4 Appendix

4.1 Hours of work

Here is how long it took to redact this document:

- Matteo Bulloni: ~ 9 hours
- Marco Cannici: ~ 12 hours

4.2 Softwares and tools used

- Google Docs: to redact the document
  Link: `http://docs.google.com`
- Lyx: to format the document
  Link: `http://lyx.org`
- Architexa plugin for Eclipse: to analyse class dependencies and generate class diagrams
  Link: `https://marketplace.eclipse.org/content/architexa-eclipse-42`

References

[1] *Transaction Service Specification*
Version: 1.4
Author: OMB - Object Management Group
Link: `http://www.omg.org/spec/TRANS/1.4/`

[2] *Java™ Transaction Service (JTS) Specification*
Version: 1.0
Author: Sun Microsystems Inc
Link: `http://download.oracle.com/otndocs/jcp/7309-jts-1.0-spec-oth-JSpec/`

[3] *CORBA Request Portable Interceptors: A Performance Analysis*
Authors: C. Marchetti, L. Verde and R. Baldoni - Dipartimento di Informatica e Sistemistica, Universita "La Sapienza" di Roma
Link: `http://midlab.diag.uniroma1.it/articoli/doa01.pdf`