



AGETOR[®]

Log & Trace
Programmers Guide

Content

1	Preface.....	4
1.1	Audience	4
1.2	Typographic conventions.....	4
1.3	Requirements	4
1.4	Acknowledgements	4
2	Introduction.....	4
3	Log API.....	4
3.1	Configuration and necessary properties.....	5
3.2	Obtaining a logger.....	5
3.2.1	Example	5
	Logging recommendations.....	6
3.2.2	URL submission of log data	7
4	Trace concepts and API	7
4.1	Transformers, documents and trace graphs.....	7
4.2	Unique document ID's and link paths.....	8
4.3	Hierarchical transformers.....	8
4.3.1	Logging as transportation mechanism for trace events.....	9
4.3.2	Transformer instances	10
4.4	Document events.....	10
4.4.1	ReceiveEvent.....	10
4.4.2	SendEvent	11
4.4.3	TransformEvent	11
4.4.4	SplitEvent.....	12
4.4.5	RelateEvent	12
4.4.6	AttachInfoEvent/AttachTraceInfoEvent	12
4.4.7	AttachTransformerInfoEvent	13
4.4.8	CheckPointEvent.....	13
4.4.9	Piggybacking properties on other events	14
4.4.10	Functions and events.....	14
4.5	Summary	16
5	Trace database and objects.....	16
5.1	Trace.....	17
5.2	TraceEvent.....	18
5.3	DocRelation	18
5.4	IOEvent.....	18
5.5	DocSplit	19
5.6	Transformer	19
5.7	Run	19
5.8	Installation.....	19
5.9	LogGroup	19
5.10	ErrorRec	19
5.11	Document	19

6	Example programs	20
6.1	Example1: ReadTransformSend	20
6.1.1	The plain Java code.....	20
6.1.2	Enriching with trace events.....	21
6.2	HierarchicalTransformers	24
6.3	HierarchicalTransformers2.....	27
6.4	SplittingSubTransformer	29
6.5	Example: communicating client/server applications	31
6.5.1	TranslatorService.....	32
6.5.2	Client code	34
6.5.3	Resulting database entries.....	36
7	Applying events to a Custom AXT Filter	40
8	References.....	40

1 Preface

This document describes how to use the AGETOR® Log & Trace systems. This includes an introduction to basic log and trace concepts and principles used by the system, an API walk-through and a number of code examples of increasing complexity.

1.1 Audience

This guide is written for developers who want to interact with the AGETOR® Log & Trace framework in order to generate log and trace events for the AGETOR® Log service.

1.2 Typographic conventions

- Text marked with *italics* refer to other publications or definitions of concepts
- Text marked like `AbstractClassName`, `identifier`, `myCoolFunction()` and `cmd` refer to executable commands, identifiers or literal code excerpts



Issues requiring your special attention are presented like this!

1.3 Requirements

You need an AGETOR® installation to use the Log & Trace framework.

1.4 Acknowledgements

The software described in this document includes software developed by the Apache Software Foundation (<http://www.apache.org>).

2 Introduction

The AGETOR Log & Trace system (L&T) is a java implementation that allows distributed event and document trace logging. The system uses the AGETOR® ORB subsystem for delivery of L&T events to a central Log Server (LS). The LS persists data in a database.

The logged events forms the basis of a centralized surveillance and notification system that allows presentation of events, their source and severity as well as the flow and transformation of documents through the AXT® business integration components.

The Log & Trace systems may be used independently; I.e. one may use the centralized logging API provided without using the Trace API and vice-versa.

3 Log API

L&T uses the Apache Commons Logging package as a thin top layer for framework independent logging. By default Log4j is the chosen and supported log framework used beneath Commons Logging. L&T uses an extension factory (`ADKLogFactory`) with Commons Logging in order to prefix loggers (names) with information relevant in a distributed context. E.g. logger names are prefixed with an application id so information from different applications may be distinguished.

A number of appenders are plugged into Log4j so that log information may be sent to different destinations. Specifically a distribution appender (`DistLog`) takes care of the ORB-based guaranteed delivery to the Log Service (LS).

3.1 Configuration and necessary properties

In order for the L&T system to work properly a number of properties should be passed to the JVM at start-up. In an ordinary AGETOR® installation these settings are correctly set on installation.

However when running your own Java programs in AGETOR® context you should provide an application id as a property *unless you give this id explicitly programmatically* when initializing loggers.

The property is given to the JVM for the L&T system to use by the argument:

```
-Dagetor.logging.appid=my-application
```

When logging or tracing, the application id will be attached to the logged information.

The remaining properties (automatically handled in an AGETOR® installation) are shown below:

```
-Dlog4j.configuration=file:/C:\projects\adk_logging\conf\agetor\logging\log4j.properties  
-Djava.util.logging.config.file=C:\projects\adk_logging\conf\agetor\logging\java_logging.properties  
-Dorg.apache.commons.logging.LogFactory=dk.bordring.inside.logging.ADKLogFactory
```

These properties specify log configuration files for Log4j, Java Logging (remapping to L&T) plus an extension factory that should be used by Apache Commons Logging for the L&T system to work properly.

3.2 Obtaining a logger

To log events a class must obtain a *logger*. The logger exists in a logger hierarchy and this hierarchy is given by its name. Typically log frameworks like *Log4j* and *Java Logging* recommends the fully qualified classname for the logger name.

3.2.1 Example

When using AGETOR® L&T Apache Commons Logging is configured to use the `ADKLogFactory` class for Log instance construction.

Obtaining a logger using the standard Commons Logging method is:

```
logger = LogFactory.getLog(TraceExample2.class);
```

If an application id has been given as a Java property and you don't need to specify a specific logical group that logging should fall under, this is all you need to get going.

In case you wish to assign a certain application id to all logging at runtime, there are two ways of achieving this: (i) attaching `appid` to running thread or (ii) explicitly give `appid` to `ADKLogFactory`. Attaching an application id to the current executing thread and all sub-threads that it spawns is done by calling the method:

```
ADKLogFactory.setAppID(String appId);
```

from the executing thread. Note that this *must* be done before obtaining the logger from the Commons log factory.

Alternatively you may go directly to the `ADKLogFactory`. The `ADKLogFactory` provides a number of static methods for obtaining a logger to use in the scope of a class instance. The factory methods allow more or less information to be given by the user. E.g. they may explicitly specify the name of the application as well as a logical log group. If not given the application id deduced from Java properties (see above).

You must always specify a logger name which should be the fully qualified name of the class that wishes to log. Methods exist for constructing this name from a simple reference to the class itself:

```
private static Log logger = ADKLogFactory.getLog(TraceExample2.class);
```

To obtain a logger for a specific name and logical log group communication you may state:

```
logger1=ADKLogFactory.getAppGroupLog(this.getClass(), "MyApp2", "communication");
```

The effect will be that log information is sent to the log group named:

```
adklog.MyApp2.communication.<classname>
```

This allows separate configuration (filtering, handling/routing) of log information to this group apart from other groups under the application.

Once a logger is obtained you may log information on a number of logging levels as shown below:

```
if(logger.isDebugEnabled()) {
    logger.debug("Now at this point in code..");
}
if(logger.isInfoEnabled()) {
    logger.info("The program was started.");
}
if(logger.isWarnEnabled()) {
    logger.warn("Memory is low");
}
if(logger.isErrorEnabled()) {
    logger.error("File could not be read");
}
if(logger.isFatalEnabled()) {
    logger.fatal("Licences expired - halting executing!");
}

try {
    throw new Exception("Exceptions was thrown!");
} catch (Throwable t) {
    logger.fatal("Exception during ... processing", t);
}
```

Note that *guard* expressions checking if the logger has the log level in question enabled are used to reduce unnecessary resource consumption.

Logging recommendations

Obtain a logger once in the lifetime of the class instance. Obtaining a new logger for each log statement made is *not* good practice.

Use guard statements to ensure that the logger has the required logging level turned on before attempting a log statement for that level. Otherwise the execution will use unnecessary resources.

If your application is a stand-alone application you may give the application id as a command line property. If, on the other hand, your application is executing in an application container (e.g.

Tomcat), several applications will co-exist and must each specify their application id to the log system.

TODO: more about thread local binding of app-id

3.2.2 URL submission of log data

For external systems log information may be submitted to the L&T system through an URL using the HTTP GET-method. The format of this URL should be that of an URL constructed by a browser from an ordinary form input with certain fields. I.e. the format would be similar to:

```
http://localhost/servlet/dk.bording.inside.logging.external.ADKLogServlet?machine=acmeserver&project=myproject&apid=hellologsystem&...
```

etc.

The required fields may be deduced from the example input form found in `agetor/logging/logentry.html`. This input form may be used from the frame page `agetor/logging/logentry.html` to submit test log submission against a running AGETOR® installation.



If you create your own URL you should ensure that it is correctly URL-encoded – i.e. spaces are converted to special escape sequences etc.



Due to limitations in naming of loggers in the Log4J framework, your machine and project names should be valid ids – they should not contain white space, periods or special characters. However you may use any unique id for a machine or project (I.e. the id need not actually be the name of the machine or project though this is probably more intuitive during later log data inspection).

4 Trace concepts and API

The AGETOR® Trace API (Trace) requires the understanding of a number of basic concepts and entities necessary for catching and reconstructing a document trace.

4.1 Transformers, documents and trace graphs

A document trace consists of an initial document event – e.g. a receive event - in an application and all transitively related documents. We shall call the process/thread/application that causes document events for the *transformer*. The *document* is a generic set of data in any representation. A virtual trace emerges as the document is exchanged with other transformers or when the document is transformed to a new document instance. This virtual trace connects the original document with all transitively related documents and effectively forms a directed *graph* [1] where documents are *nodes* and exchanges, transformations and other types of relations are *edges*.

The most basic transformation is converting a document to another data set. A more complex example is the splitting of a document into several other documents. The base line is that the original data set is related to one or more data sets and this relation is itself related to the transformer performing the relational actions.

The AGETOR® trace API may be used for generating document relating events that allows the reconstruction of the virtual graph as an actual visual graph or alternatively a related set of exchange and transformation events in a textual presentation. Figure 1 illustrates how such document flow could be visualized. Yellow squares represent documents (nodes), arrows (edges) illustrate send, receive, transforms and splits of documents and the surrounding nested boxes

illustrates hierarchical organized transformers and application contexts. As can be seen the exchanged documents form a graph with a hierarchy of transformers superimposed (the transformer hierarchy may also be viewed as a way of defining sub-graphs).

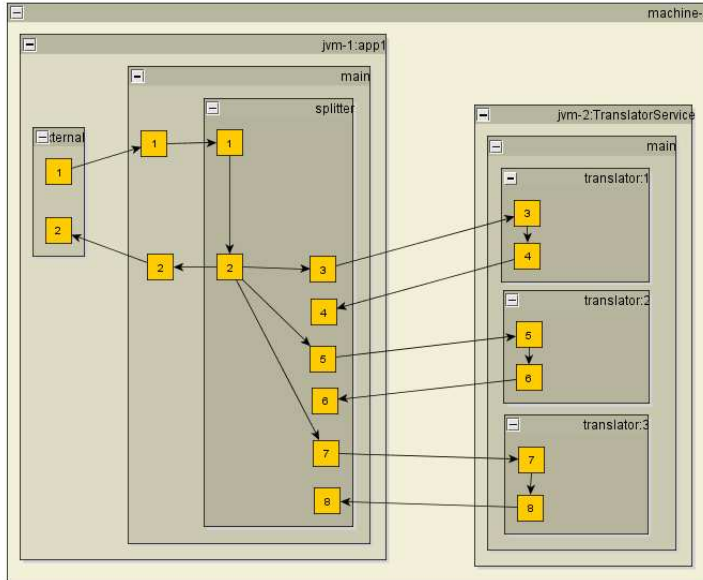


Figure 1. Transformers sending, transforming, receiving documents

4.2 Unique document ID's and link paths

Documents that are to be traced must be assigned globally unique ids. When documents are exchanged across applications and machines (sent, received) the document id *must* travel with the document data set in order for the receiving transformer to relate the receive event with the send event (binding together the two documents in the graph).

The Trace API allows the construction of unique document id instances (aka DID's) . These encode the machine name, the JVM-id and a sequentially increasing number and are thus *globally unique*. The class DID contains this information and instances are simply created with

```
DID did = new DID();
```

Since objects cannot easily be exchanged in distributed environments, a string representation of the DID is used rather than the object itself. The string representation is obtained with

```
String didString = did.toString();
```

and this string contains the machine, JVM-id and the number in a Base64 [2] encoding.

To uniquely register the traversed path of exchanged documents, a *path* string may be piggybacked with the document string. This path is encoded into the document string by the `SendEvent` class as we shall see later.

4.3 Hierarchical transformers

Transformers are hierarchically structured such that one transformer may be the parent of multiple child-transformers. With respect to the Trace API, transformers are merely an abstract concept in

that the transformer instances used for generating event do not perform any transformation, exchange or document splitting for that matter. All the actual document processing and exchange is the responsibility of the application that uses the Trace API and the Trace API merely records the events. Thus one could generate events for constructing graphs showing send/receive/transform of documents without actually doing any document sending or transformation!

A root transformer may be obtained by a statement like:

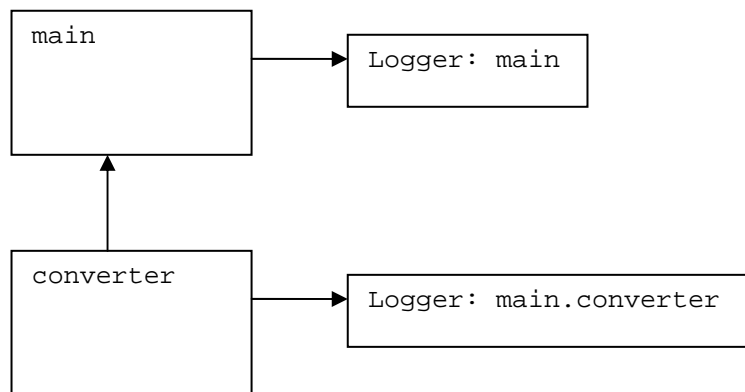
```
TID rootTID = TraceFactory.createRootTID(ADKLogFactory.getAppID(), "main");
```

The transformer `rootTID` is given the type name "main" and is registered with the application id returned from the `ADKLogFactory` by the `getAppID()` method.

Now a child transformer could be constructed with the method `createChildTID()`. The method takes the name of the child transformer.

```
TID tid = parent.createChildTID("converter");
```

Structurally this creates the following relation:



I.e. the child (or *sub-transformer*) relates to its parent transformer (has a reference). The parent transformer will use the logger named `main`, whereas the child uses the logger called `main.converter` (see below).

4.3.1 Logging as transportation mechanism for trace events

The `TID` class is representing a transformer node in the transformer tree. It wraps a `Log`-instance which it uses for logging document trace events (all information about the event is encoded into a string which is sent as a log message). The hierarchical transformer naming is directly used as logger name. Normal application name and logical group prefixing is applied under the hood such that trace events spawned from the child transformer constructed above would go to the logger named `adktrace.myapp.none.main.converter` (if we assume that the application name is `myapp`). The default group used is `none`.

4.3.2 Transformer instances

Whenever a transformer is constructed it gets a *transformer instance name* which is unique throughout the run of the JVM. The instance name has a hierarchically structure corresponding to the name of the transformer but with the difference that each level of the hierarchy is a unique number. I.e. the multiple calls creating a transformer results in different instances:

```
TID tid1 = parent.createChildTID("converter");
TID tid2 = parent.createChildTID("converter");
```

Produces instances with naming:

```
1.1
1.2
```

for the two transformers. Since the parent transformer is the same the leftmost 1 is the same. For each child an increasing number indicates a new instance. The Trace system only requires that a new transformer gets a unique instance number for its “level” in the instance hierarchy whereas the instance prefix must be the same for two children of the same parent transformer.

You may think of the transformer name, e.g. `main.converter`, as the type of the transformer whereas the instance names, e.g. `1.2` or `1.7` identify instances of the type.

4.4 Document events

Document events are generated from transformers. The transformer class (TID) contains factory-methods that returns new instances of the events and internally relates the events to the transformer.

As mentioned, all trace events results in log events using the `Logger` instance of the transformer. The log message text is used for encoding the trace information into a string representation that is decoded in the Log Server and persisted in dedicated Trace tables.

Internally the trace event classes all hold a set of properties (Java Property class) with all required event information held as name/value pairs. The set of properties is simply serialized into string representation and sent as the log text message.

All the document events follow a simple pattern:

```
e = tid.createXXXEvent()
try {
    // do the document processing
    e.commit();
} catch (Exception ex) {
    e.fail("couldn't process document", ex);
}
```

The events differ slightly in what parameters are given to the commit method and what other methods are available. The following sections detail the available event types.

4.4.1 ReceiveEvent

The `ReceiveEvent` is used for registering that a document is received. The document may stem from a source that did not construct a document id for the document in which case one must be constructed at the time of receive.

If, on the other hand, a document id string is received through the receive channel with the document, it is important that the receive event is generated for the document id string received.

4.4.1.1 Usage

The example below illustrates how the receive event is generated having a document id string (`src_did`):

```
TID tid = parent.createChildTID("converter");
// generate a receive event
ReceiveEvent re = tid.getReceiveEvent();
re.commit(src_did);
```

4.4.2 `sendEvent`

Generates a send event. This event is special in that an input document id string must be given and a resulting document id string *must* be obtained from the class. This is because the document to be sent may have encoded path information that should be extended with further path information.

When a document id is first created with the new `DID()` method, a zero-length path string is encoded into the document string. When the document is sent, a new unique path suffix is appended to the string thus generating a path like:

.1

This path is encoding into the document string and thus not directly visible in the base64 encoded string. When the document string is received by another transformer, this transformer may want to send the document on and in order to distinguish between document received/sent thru differing paths, the document string is suffixed with a new unique path number – e.g.:

.1.17

which is encoding into a resulting document id string. The example below illustrates how a send event is created using a received did string. The resulting did-string is obtained with the `toString()` method which encodes a new extended path string into the document id string.

```
// now send the result and register the sending
SendEvent se = tid.getSendEvent(receivedDIDString);
String sndDID = se.toString();
try {
    // the actual send. The DID is carried to the receiver somehow
    send(sndDID, out);
    se.commit();
} catch (Exception e) {
    se.fail("Error sending document", e);
}
```

4.4.3 `TransformEvent`

The transform event accepts a source did string as input and is able to generate a resulting did string with the `createResultDID()` method.

```
TransformEvent te = tid.getTransformEvent(src_did);
try {
    // transform input to output
    transform(in, out);
}
```

```
// ok, the transform went well we register this
te.commit(te.createResultDID());
...
...
```

4.4.4 SplitEvent

See source examples later.

4.4.5 RelateEvent

Relates one document to another effectively forcing them to belong to same document trace. The example below illustrates how two documents (contained in the `dpair` object) are explicitly forced to be related using the `RelateEvent` obtained with the `getRelateEvent()` method of a transformer. The compact code example obtains, and commits the event in one line.

```
/**
 * Explicitly relate the input document with the generated
 * error output document so they are registered for same trace.
 */
tid.getRelateEvent(dpair.getSource()).commit(dpair.getResult());
```

The typical usage is whenever a document emerges that is not related to another document directly by send/receive or through transform/splitting. For example if an email is sent in response to some document transformation then one may want to register the sent document as related to the transformed document.

4.4.6 AttachInfoEvent/AttachTraceInfoEvent

Attaches information about a specific document or trace to the document/trace. The attached information is simply name/value properties that are stored with the document/trace in the database.

In Listing 1 a transform is made from one document to another. This is followed by an `AttachInfoEvent` on the result (line 75). In the subsequent lines three document properties are set on the result document. The first two are `type` and `ref(ERENCE)` which have convenience methods on the event. The third property is a user defined property (“manual-processor”) thus both name and value is given to the `setProperty()` method.. The final `commit()` sends the properties to the server.

The following code sequence attaches properties to the Trace that the resulting document is part of. Thus any document part of the trace could have been used as argument to the `getAttachTraceInfoEvent()`.

```

64     TransformEvent te = tid.getTransformEvent(src_did);
65     try {
66         // transform input to output
67         transform(in, out);
68         // ok, the transform went well we register this
69         te.commit(te.createResultDID());
70         // also set the result of this transformer in the
71         // dpair as result for the caller to use
72         dpair.setResult(te.getResult());
73         // Simulate extraction of document info and registration of
74         // document type and reference thru AttachInfoEvent
75         AttachInfoEvent ae = tid.getAttachInfoEvent(te.getResult());
76         ae.setDocType("order batch");
77         ae.setDocRef("order 3444323-1");
78         ae.setProperty("manual-processor", "bbo");
79         ae.commit();
80
81         AttachInfoEvent ae2 = tid.getAttachTraceInfoEvent(te.getResult());
82         ae2.setProperty("name", "Acme Inc. order processing");
83         ae2.setProperty("priority", "medium");
84         ae2.commit();
85         // now send the result and register the sending
86         SendEvent se = tid.getSendEvent(te.getResult());
87         String sndDID = se.toString();
88         try {
89             // the actual send. The DID is carried to the receiver somehow
90             send(sndDID, out);
91             se.commit();
92         } catch (Exception e) {
93             se.fail("Error sending document", e);
94         }

```

Listing 1. Attaching info to trace and documents

4.4.7 AttachTransformerInfoEvent

Yet to come

4.4.8 CheckPointEvent

The CheckPointEvent is used for displaying useful information to the end user about the current state of the transformation process. The information supplied with a checkpoint event is displayed directly to the user. Furthermore the description for checkpoint is stored in a separate database table and thus has no relation with other events whatsoever. The only thing that binds a checkpoint to a trace is the document it references.

Unlike other events a checkpoints sole purpose is information/logging and therefore it has a separate level determining the importance of the information to display. These levels are equivalent to the ones found for log messages i.e. (DEBUG,INFO,WARNING,ERROR,FATAL). Default is info.

4.4.8.1 Usage

The example below illustrates how a checkpoint event is generated having a document id string (src_did):

```

CheckPointEvent cp = tid.getCheckPointEvent(src_did); // create a checkpoint event
cp.setLabel("Transformed input to UPPERCASE");      // set the checkpoint label/description

```

```
cp.setLevel("INFO"); // set the checkpoint level
cp.commit();
```

4.4.9 Piggybacking properties on other events

Though explicit attach events exists, one may piggyback document properties on any of the other events. The `setProperty()`, `setDocType()`, `setDocRef()` and `setDocExternalRef()` methods are all present on the other event types. For events involving one single document the properties pertain to that document (e.g. send, receive). For events involving two documents (relational events as transform, split, relate) the properties pertain to the “to” document (result/target). Piggybacking document properties result in better performance than using two individual events. See example section for a piggybacking example.

Trace properties may not be piggybacked.

4.4.10 Functions and events

The L&T API support a simple function abstraction that allow the user/applications to relate functions to trace events and documents. The functions simply resolves to URL-strings in the L&T GUI and these URL should provide the functionality in the shape of e.g. a web-page that could manipulate a document in kind of way. I.e. the implementation of the web-page is completely the responsibility of the application developer that creates the functions.

Thus the function abstraction is a way to extend the GUI of the L&T system in a context-sensible way. The GUI allow client applications to register the URL information that will open a page on a remote machine.

Functions may be attached either as part of another document event or by explicit stand-alone events referring to a previous event or document. The latter requires more communication but allows attachment of multiple functions to an event or document. It also allows for the attachment to be postponed in time with respect to the original event.

At present the AGETOR AXT system attaches functions for transformation events that allow the user to open a web-page that details the error and allows for download of the failed input-data. Similarly the DDS-server attaches a redelivery-function when a document delivery fails. This redelivery function (URL) opens a redelivery-web-page from which the user is able to inspect/modify delivery properties for a document as well as up/download the document data for local inspection/modification.

4.4.10.1 The FunctionHolder class

This class is used for creating the function that should be attached to a document or event. The class accepts the following parameters

- URL base
- function name
- a parameter set

The class is used in connection with the function attach event described in the following sub-sections.

4.4.10.2 Attaching functions using other document events

The document events allow the attachment of one document function and one event function. This is achieved by calling the `setEventFunction()` and `setDocFunction()` on the document event passing a `FunctionHolder` instance as argument.

The example below shows how a document function is piggybacked on a send event. This has the effect that the function is attached to the source document of the send event. In the example the relative URL `AXT/dds` is given as URL base. The function name is `download`. This results in an URL in the L&T system with the value: “`http://<lt-hostport>/AXT/dds/download?docid=dds/<jobid>`” where `<jobid>` is the result of the `getId()` function below.

```
SendEvent se = tid.getSendEvent(cfg.getDID(), "error-repository (" + newPath + ")");
fh = new FunctionHolder("AXT/dds", "download");
fh.setParameter("docid", "dds/" + jobInfo.getStatus().getId());
```

If the URL base is absolut (e.g. `http://<server>:<port>/path`) then this absolute value will be used of course. In the L&T GUI the function will be available and indicated by a function icon in the user interface.

The next example is similar but attaches a function to the send event itself.

```
FunctionHolder fh = new FunctionHolder("AXT/dds", "redelivery");
fh.setParameter("docid", "dds/" + jobInfo.getStatus().getId());
se.setEventFunction(fh);
```

4.4.10.3 AttachEventFunctionEvent

This event allows the attachment of a function to an event. The event to which the function is attached has already previously been submitted and in order to refer to the previous event you should pass the event id.

In the example below a function is constructed in the first two lines. In line 3 the event attaching the function to the previous event (`se`) is created. The arguments are the document id that was used for the previous event (as source), the previous event and the function.

```
1. fh = new FunctionHolder("AXT", "laterFunction");
2. fh.setParameter("later-did", cfg.getDID());
3. AttachEventFunctionEvent aef = tid.getAttachEventFunctionEvent(cfg.getDID(), se, fh);
4. aef.commit();
```

It is possible to avoid keeping the reference to the previous event and only keep its internal event number (a string). This is illustrated below where the reference number is taken from a send event (`se`) and later used in `getAttachEventFunctionEvent` method as an argument rather than the event itself.

```
5. String ref = se.getEventNo();
6. FunctionHolder fh = new FunctionHolder("AXT", "xfmerr");
7. fh.setParameter("myparam", "myvalue");
8. tid.getAttachEventFunctionEvent(did, ref, fh).commit();
```

4.4.10.4 AttacheDocumentFunctionEvent

This event is similar to the `AttachEventFunctionEvent` but only needs the `did` of the document and a `FunctionHolder` reference.

4.5 Summary

- Traces are graphs where documents are nodes and events (send, receive, transform etc.) are edges
- All documents have globally unique id's
- Trace events generates data allowing the reconstruction of the trace graph (for visualization)
- Document events are generated by *transformers*
- Transformers uses the AGETOR® log system for guaranteed firewall indifferent event transport
- Transformers may be nested creating a transformer hierarchy
- Transformers have a hierarchical names (aka transformer type)
- Multiple instances of a transformer may be created each having a unique instance id
- The events from a sub transformer correspond to a sub graph in the trace graph
- Send/Receive events serves to illustrate exchange of documents
- Transform events serves to illustrate document transformations (one-to-one relate result to source)
- Split events serves to relate document fragments to original document (one-to-many)
- Attach events serves to relate simple name/value pieces of information to either document instance or traces (e.g. an order number)
- CheckPoint events serves to ease the understanding and visualization of a trace.

5 Trace database and objects

The document and trace events are persisted in the Log database (through Log Server). When the event arrives (as a textual string in a log message), the LS splits it out into a number of tables. These tables have been normalized to reduce space complexity and redundancy.

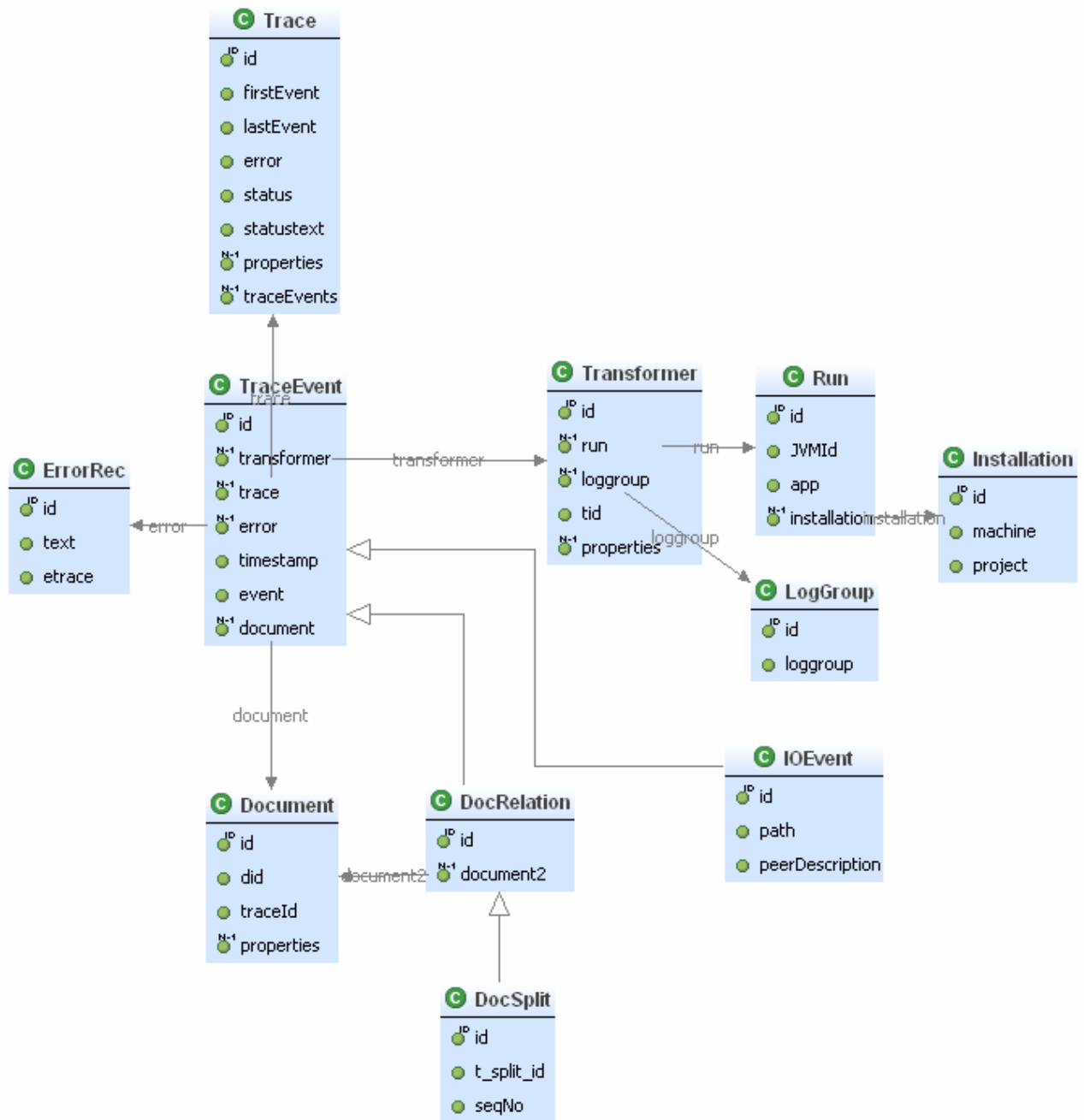


Figure 2. Hibernate Trace objects used by Log Server

Corresponding to these structures are a number of Java Objects (used with Hibernate) that support automatic association and lazy load of associated objects (Figure 2). E.g. the `TraceEvent` structure has a reference to the `Trace` to which it belongs, while the `Trace` holds a list of references to the related `TraceEvents`.

5.1 Trace

All events that are related through the exchange of the same documents end up belonging to the same `Trace`. Sometimes events arrive to the Log Server out of order from many sources (e.g. 2-3

different machines) even though they are part of the same document flow. The Log Server seeks to deduce the trace that events belong to. At some points the lack of one “binding” event (e.g. a transformation) may cause the server to create two distinct Traces until the missing event arrives later. Then the two Traces are merged into one.

For each trace deduced, a Trace record is made. The trace record holds the time interval of its related trace events. From the trace object a list of all trace events may be obtained. These are lazy loaded by Hibernate on request.

The Trace record also holds any trace properties attached during program execution (using the Attach event type)

5.2 TraceEvent

The TraceEvent object is the central object of the trace database. One record exists for each send, receive, transform, split and relate event generated by programs. The TraceEvent has the following fields:

Field	Description
Transformer	A reference to the Transformer instance that generated the record
Trace	A reference to the Trace object.
Error	A reference to an Error record (if any)
Timestamp	The time that this event was generated
Event	The type of event. This is a one char indication. s = send r = receive c = checkpoint t = transform S = split R = relation If the event was error-prone, the type is prefixed with an e for error. E.g. es, er, et, eS.
Document	A reference to the (source) Document object of this event. Some events have a resulting or related object (transformation, split, relation) that are stored in the sub-classes: DocRelation, DocSplit
tid	The transformer instance id. This important piece of information reveals how the transformer that generated the event is related to parent transformer instances.

5.3 DocRelation

This is a sub-class of the TraceEvent. It extends the trace event with information about a related document. If the base event type is transform (t) then this object simply holds the resulting document id of the transform.

If the event type is R (relation), this object holds the document id of the related document.

5.4 IOEvent

This is a subclass of the TraceEvent. It is used whenever the event type is Send or Receive. The object holds the path (see 4.4.2 for description) of the event and this information should be used for binding together related send and receive events when visualizing the flow. I.e. the transformer

instance generating an send event with a specific path for a specific document have a “link” to the transformer receiving the same document instance with the exact same path.

Besides, a plain text description/id of the peer to/from the document was sent is present (`peerDescription`). This description should be used when no corresponding `IOEvent` can be found in the trace since it gives a textual information about the source or destination (e.g. “*file system*” or “*FTP-server ftpserver.acme.com:21*”).

5.5 DocSplit

This sub-class of `TraceEvent` is used when the event type is `S` (`SplitEvent`).

Field	Description
<code>T_split_id</code>	The id of the split within the transformer that made the split. I.e. if the same transformer does multiple splits of different documents, all events of the same basic split will have the same <code>t_split_id</code> (The id is assigned by the <code>SplitEvent</code> class when it is constructed).
<code>seqNo</code>	The ordering on documents split from the same base document.

5.6 Transformer

The transformer record holds a reference to the `Run` and the `LogGroup`.

Besides it holds properties of the specific transformer instance (if any). Particularly the “desc” property is used for attaching transformer information/description by users and have this visualized in the GUI as part of transformer names.

5.7 Run

The `Run` record holds the application ID, JVM instance and a reference to the `Installation` on which the run was made.

5.8 Installation

Holds the machine and project name of the `Trace` event.

5.9 LogGroup

The log group is equivalent to the logger or transformer name. It is the hierarchical (dot-separated) name of the logger/transformer that made the event. E.g. if a sub-transformer of the main-transformer called `splitter` generates an event, the log group will be `main.splitter`.

The name may be viewed as the transformer type whereas the `tid` field of the `TraceEvent` records is the instance. Many instances may exist of the same type.

5.10 ErrorRec

This object holds an error (if any) and may contain an error text and a more elaborate exception trace.

5.11 Document

One `Document` object exists for each unique document found from traces. Note that multiple send/receive events of the same document with different paths does not result in more documents.

The document object holds the document properties that were attached by programs (using `Attach` events) during their execution.

6 Example programs

This section gives examples on how to use the Trace API for registering document flow events. A number of programs exercising increasing complexity are provided.

6.1 Example1: ReadTransformSend

This example illustrates how a program may register receive, transform and send events for documents. The program reads a file from disk, transforms the content and store the file back. This corresponds to the three event types: receive (from file system), transform and send (to file system).

All events are made from one root transformer for simplicity. Later examples will show the usages and document exchange with more transformer levels.

6.1.1 The plain Java code

Since we wish to accept input from `stdin` and subsequently write output to `stdout`. Redirection could be used from a prompt to have input taken from a file and sent to an output file.

The plain Java code is show below:

```
public class ReadTransformSendPlain {

    public void runExample() throws IOException {
        /**
         * Read the file content to a buffer
         */
        String inStr = null;
        String outStr = null;

        BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
        StringBuffer sbuf = new StringBuffer();
        String line;
        while ((line = br.readLine()) != null) {
            sbuf.append(line+"\n");
        }
        inStr = sbuf.toString();
        /**
         * Transform to uppercase
         */
        outStr = inStr.toUpperCase();
        /**
         * Write back the transformed content to a file
         */
        System.out.print(outStr);
    }

    public static void main(String[] args) throws CommandLineException, IOException {
        ReadTransformSendPlain ex = new ReadTransformSendPlain();
        ex.runExample();
    }
}
```

6.1.2 Enriching with trace events

The three main functions of the program is to read (receive), transform and write result (send) the data. We shall now enrich the code with Trace events that will register these events and create appropriate document ID's. The example uses only one transformer (main) for simplicity.

6.1.2.1 Transformer creation and the receive event code

```
30 public void runExample() throws Exception {
31     String inStr = null;
32     String outStr = null;
33     /**
34      * Obtain a root transformer instance and name it 'main'.
35      */
36     TID tid = ADKTraceFactory.createRootTID(ADKLogFactory.getAppID()
37         , "main", null);
38     /**
39      * Read the file content to a buffer
40      */
41     // Create a new receive event object. Also create a brand new
42     // document ID since we dont receive one with the input.
43     ReceiveEvent re = tid.getReceiveEvent();
44     String docId = new DID().toString();
45
46     try {
47         BufferedReader br = new BufferedReader(new InputStreamReader(
48             System.in));
49         StringBuffer sbuf = new StringBuffer();
50         String line;
51         while ((line = br.readLine()) != null) {
52             sbuf.append(line + "\n");
53         }
54         inStr = sbuf.toString();
55         re.commit(docId);
56
57     } catch (Exception e) {
58         re.fail("Unable to get input, e");
59         throw e;
60     }
```

The code snippet above shows how the reading of input is enriched with Trace Receive events. In line 36 the root transformer instance is created. In line 43 a new receive event object is made. We use this to register when we successfully receive a document or have an error while receiving a document.

Since the “document” is a number of text lines obtained from `stdin` there is no existing document ID for this input. Thus we create a new document ID in line 44. This ID must be used when we register the successful receipt of a document. Normally this will arrive *with* the document through the communication channel but not in our simple example.

In lines 47-54 the reading and in-memory storing of data is done as before. We have surrounded the I/O with a try/catch construct so we can handle success/failure by either commit'ing the event or fail'ing it.

The snippet above will send a receive event to the backend system with information on the transformer and document ID involved. The event will be part of a trace containing all events that are related by sending/receiving or transforming.

6.1.2.2 Transformation event code

The next section of code shows how the transformation is registered.

```
61      /**
62       * Transform to uppercase
63       */
64      // create a transform event
65      TransformEvent te = tid.getTransformEvent(docId);
66      try {
67          // The actual transform
68          outStr = inStr.toUpperCase();
69          // commit with a new result id
70          te.commit();
71      } catch (Exception e) {
72          te.fail("Unable to transform input string", e);
73          throw e;
74      }
```

In line 65 a transformation event object is made. As argument we give the document ID that we created previously. This is important since we wish to register the transformation of the data represented by that document ID. Again we use the try/catch and commit/fail construct and abstraction. The `TransformEvent.commit()` automatically creates a new document ID for the result that may be obtain subsequently if necessary (see below).

The effect of this event is that the original and new document ID's are registered as related by transformation in the backend database. Thus they become part of the same overall trace.

6.1.2.3 Send event code

Finally lets see how to register the send of a document.

```
75      /**
76       * Write back the transformed content to a file
77       */
78      // Create a send event object and use the resulting document
79      // id from the previous transformation
80      SendEvent se = tid.getSendEvent(te.getResult());
81      try {
82          System.out.print(outStr);
83          se.commit();
84      } catch (Exception e) {
85          se.fail("Could now write back result", e);
86          throw e;
87      }
```

In line 80 the resulting document ID from the previous transformation is given as the ID of the document we are sending (physically sending to a file/stdout). Illustration 1 shows how the transformer (main) is related to the events that are related to document ID's (1,2 for simplicity).

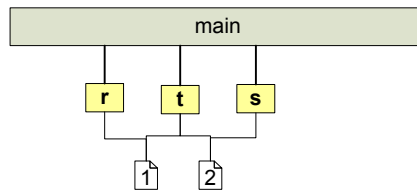


Illustration 1: One transformer and tree events binding documents together in a trace

Besides the transformer and events are registered with information about the ID of the running program, the machine, project, JVM instance and other information necessary to present traces subsequently.

An abstract depiction of the exchange is given in Figure 3. It shows how the main transformer receives the document denoted 1 from an external source. The document is transformed to document 2 which is sent to the external source. Such a depiction could be created from the send/transform/receive events that are involved in conjunction with the used transformer naming. In lack of events generated from the external sources (i.e. there is no transformer who have actually sent document 1) a program could simply create a dummy transformer denoted external or similar and depict that the document was sent from there.

We shall use this kind of visualization for the following examples as well since they present the document flow and transformer hierarchy in an easy readable and intuitive manner.

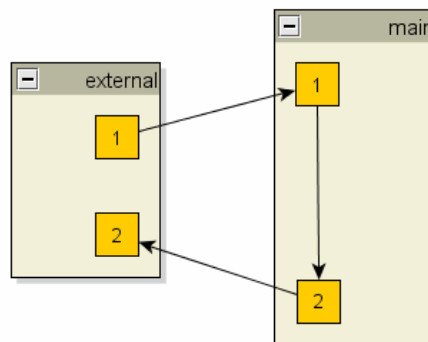


Figure 3: Abstract graphical depiction of document exchange

The main transformer executed in an application instance in a specific JVM-instance on a specific machine. One may think of the complete execution context as being a higher level container or transformer in which the main transformer is contained. We could illustrate this as in Figure 4 where `<m>:<jvm>:<app>` would be replaced with actual machine, JVM-ID and application name.

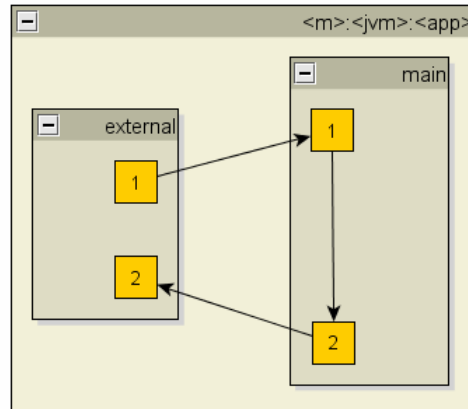


Figure 4: Execution context depicted as a "mother" container of transformer

6.2 HierarchicalTransformers

In applications where complex nested processing takes place it is required that we are able to catch the nested layers of processing. E.g. a main processor may want to delegate parts of the processing to a sub-transformer/processor. The motivation behind capturing the processing hierarchy and division is that this allows a subsequent presentation of the actual processing and allow that parts of the processing is collapsed/expanded to hide/see details. E.g. a higher level processor could show its event while all sub-transformer events were hidden under an expandable node.

We both need to catch the nesting of transformers but also the exchange of documents going on. The figure Figure 5 below shows our processing from before but now with the transformation being done in a sub-transformer named `UCTransformer`.

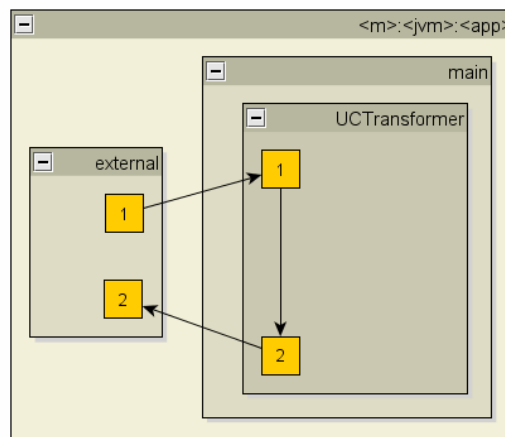


Figure 5: Sub-transformer does transform

To achieve the information capture needed for this illustration a program must simply create and use a sub-transformer for the transform event.

The next program is divided into two classes. The main program (`HierarchicalTransformers`) and a class that does the "sub"-transformation (`SimpleSubTransformer`).

The example code is the same as before: the main program reads a file/stdin and registers the Receive event. The transformation however is now delegated to the `SimpleSubTransformer` which registers the Transform event. The input/output document is passed using `StringBuffer` objects. The main program finally registers the Send event as before. There is no reason that a sub-transformer event should be in a distinct class – we only do this to illustrate the division of work and it may often be the case.

Note that the main program creates a `DPair` object that simply holds a pair of document ID's. This is used for exchanging the ID of the resulting (transformed) document since this ID is created by the sub-transformer but required by the main transformer which must register the sending of the transformed document.

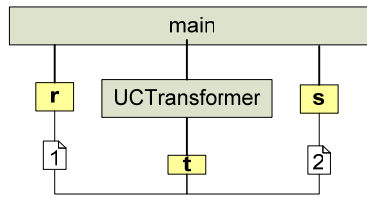
The main (parent) transformer is also passed as a parameter to the sub transformer in order for the sub transformer to be able to create its own sub-transformer which is a child of the main-transformer. The `createChildTID()` method in line 19 is used for this purpose.

```
7 public class SimpleSubTransformer {
8
9
10 /**
11  *
12  * @param parent Parent transformer in the hierarchy
13  * @param docIDPair a holder object that can hold input and output document id
14  * @param input the data to transform
15  * @param output the buffer for the result
16  * @throws Exception
17  */
18 public void transform(TID parent, DPair docIDPair, StringBuffer input, StringBuffer output)
19     TID tid = parent.createChildTID("UCTransformer");
20     /**
21     * Transform to uppercase
22     */
23     // create a transform event
24     TransformEvent te = tid.getTransformEvent(docIDPair.getSource());
25     try {
26         // The actual transform
27         output.append(input.toString().toUpperCase());
28         // commit with a new result id
29         te.commit();
30         // set the id of the transformed result for the parent to use..
31         docIDPair.setResult(te.getResult());
32     } catch (Exception e) {
33         te.fail("Unable to transform input string", e);
34         throw e;
35     }
36
37 }
38 }
```

Figure 6: simple sub transformer in its own class file

```
33 public class HierarchicalTransformers {
34
35     public void runExample() throws Exception {
36         /**
37          * Obtain a root transformer instance and name it 'main'.
38          */
39         TID tid = ADKTraceFactory.createRootTID(ADKLogFactory.getAppID()
40             , "main", null);
41         /**
42          * Read the file content to a buffer
43          */
44         // Create a new receive event object. Also create a brand new
45         // document ID since we dont receive one with the input.
46         ReceiveEvent re = tid.getReceiveEvent("stding");
47         String docId = new DID().toString();
48
49         StringBuffer inbuf = new StringBuffer();
50         StringBuffer outbuf = new StringBuffer();
51
52         try {
53             BufferedReader br = new BufferedReader(new InputStreamReader(
54                 System.in));
55             String line;
56             while ((line = br.readLine()) != null) {
57                 inbuf.append(line + "\n");
58             }
59             re.commit(docId);
60
61         } catch (Exception e) {
62             re.fail("Unable to get input, e");
63             throw e;
64         }
65         // create a doc id pair holder and set the id of the input document
66         DPair docIDPair = new DPair(docId);
67         // delegate the transformation to a sub transformer.
68         new SimpleSubTransformer().transform(tid, docIDPair, inbuf, outbuf);
69         /**
70          * Write back the transformed content to a file
71          */
72         // Create a send event object and use the resulting document
73         // id from the previous transformation
74         SendEvent se = tid.getSendEvent(docIDPair.getResult(), "stdout");
75         try {
76             System.out.print(outbuf.toString());
77             se.commit();
78         } catch (Exception e) {
79             se.fail("Could now write back result", e);
80             throw e;
81         }
82     }
}
```

The events may be illustrated as before with the transform-event now related to the sub-transformer.



6.3 HierarchicalTransformers2

Though the previous program captures the processing hierarchy, the information about how document 1 is passed from `main` → `UCTransformer` and from `UCTransformer` → `main` is not directly captured. With the receive/transform/send events used we would not know with certainty that this exchange took place.

To explicate the exchange between `main` and sub-transformer we may insert additional send/receive events in the two classes. I.e. we should have the `main` transformer “send” document 1 to the sub-transformer which should “receive” it. Similarly the sub-transformer could “send” document 2 which could be “received” by the `main`-transformer.

Doing this will reconstruct the document exchange path subsequently from the registered information.

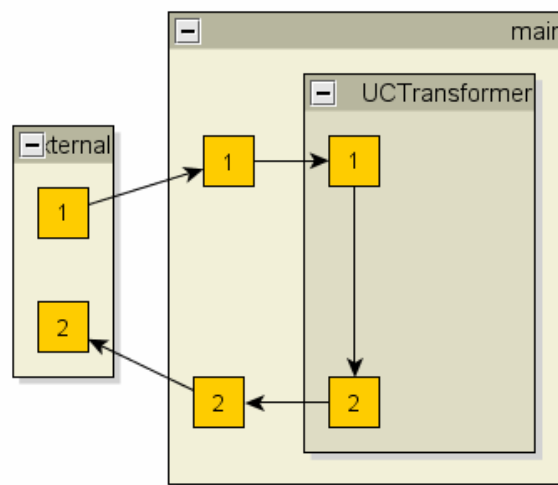


Figure 7: Explicit send/receive events between transformers to capture flow

The required modification to the `main` program can be seen in Listing 2. The only difference from previous example is that we have now inserted a `Send`-event before the sub-transformer is called upon (line 64, 70) and a `Receive` event to register the return of the document (line 76, 77). The called-upon (line 72) modified sub-transformer implementation is called `SimpleSubtransformer2` (listed later).

Remember that all `Send` events result in a result document ID that is different from the original ID of a document. The `send` event encodes a piece of linking or path information that allows

tracking the document. This allows for the same document (with the same basic ID) to be sent/received multiple times all with distinct paths encoded as part of the document ID since the sender and receiver registers an event for the same document and path. For the very same reason it is crucial to hand the result of the `SendEvent.toString()` method as document ID to the transformer that makes the corresponding `ReceiveEvent`.

```
61     /**
62     * Register sending the read document onwards to another transformer
63     */
64     >> SendEvent se = tid.getSendEvent(docId);
65     // create a doc id pair holder and set the id of the sent document
66     // It is important to use the toString() method on the SendEvent
67     // to get the id generated for sending since this id encodes a
68     // special link number that connects sent/received documents.
69     DPair docIDPair = new DPair(se.toString());
70     se.commit();
71     // delegate the transformation to a sub transformer.
72     new SimpleSubTransformer2().transform(tid, docIDPair, inbuf, outbuf);
73     /**
74     * Register the recivement of the transformed document from the sub-transformer
75     */
76     >> re = tid.getReceiveEvent();
77     re.commit(docIDPair.getResult());
```

Listing 2: send/receive events in main transformer before/after sub-transformer

The sub-transformer is modified to generate Receive/Send event corresponding to the Send/Receive events of the main transformer.

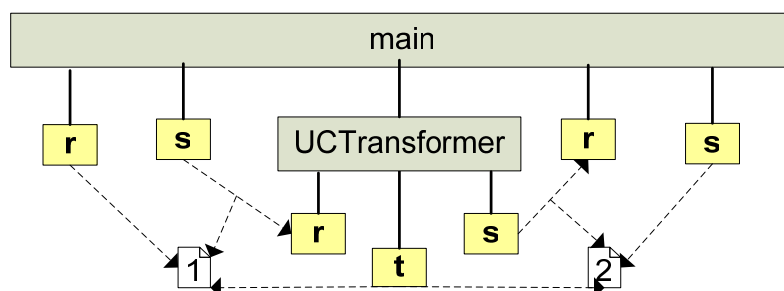
```

20 public void transform(TID parent, DPair docIDPair, StringBuffer input, Str
21     TID tid = parent.createChildTID("UCTransformer");
22
23     /**
24      * Register receiving the dataset
25      */
26     ReceiveEvent re = tid.getReceiveEvent();
27     >> re.commit(docIDPair.getSource());
28
29     /**
30      * Transform to uppercase
31      */
32     // create a transform event
33     TransformEvent te = tid.getTransformEvent(docIDPair.getSource());
34     try {
35         // The actual transform
36         output.append(input.toString().toUpperCase());
37         // commit with a new result id
38         te.commit();
39     } catch (Exception e) {
40         te.fail("Unable to transform input string", e);
41         throw e;
42     }
43
44     /**
45      * Register sending the dataset
46      */
47     >> SendEvent se = tid.getSendEvent(te.getResult());
48     se.commit();
49     // We convey the id obtained with the send event (which is
50     // different from the transform event result (encoding a
51     // link path)
52     docIDPair.setResult(se.toString());
53
54

```

Listing 3: Explicit R/S events in sub transformer

The complete set of events generated could be illustrated as below. Two transformers registers a series of events on two document instances.



6.4 SplittingSubTransformer

The next example shows how to generate events corresponding to the splitting of one document into multiple other documents. The split is implemented in the sub-transformer after the transformation has taken place. I.e. we split the transformed result into a number of fragments. In this case it is

necessary to create events that relate the original document (ID) to multiple new documents (IDs). We shall also illustrate how to attach pieces of information to each fragment.

As for other events an object corresponding to the event is created. This object is of class `DocumentSplitEvent`. This object is different from the previous types since it allows for multiple `commit()` calls once created. Each `commit()` call corresponds to a new Document fragment event.

We have modified the `SimpleSubTransformer` to `SplittingSubTransformer` which split the transformed document from previously (Remember that the sub-transformer simply uppercased the input document). The example split that is done will simply split the document into a new “document” for each line present in the input (i.e. `\n` separated text lines).

The additional code is shown below. First we create a `DocumentSplitEvent` which is given the document ID of the document which is being split. In our case this is the output from the uppercase-transform. In line 48 this is done.

A loop around the lines of the input is made in line 52-63. For each iteration (i.e. new line in the input) we reset the split event object first (`dse.reset()` in line 55). This clears any state information from the previous iteration (any attached information). We then make a brand new document ID for the new document fragment (the extracted line) and *attach* some document information (line 57-61). Line 59-61 actually has nothing to do with the splitting registration but only serves as example for how it is possible to attach pieces of information to a document using the `setDocType()` and more generically `setProperty()` methods. When using the split event the information is attached to the *resulting* document rather than the original. Note that `setDocType(typeString)` is equivalent to calling `setProperty("type", typeString)`.

In line 63 the document split (including the attached information) is committed to the back-end server.

```
45     /**
46     * The input has been uppercased. Now we split it into lines.
47     */
48     DocumentSplitEvent dse = tid.getDocumentSplitEvent(te.getResult());
49     try {
50         String[] lines = input.toString().split("\n");
51         String [] dids = new String[lines.length];
52         for(int i=0; i<lines.length; i++) {
53             // clear the held data in the splitter if any from
54             // previous split operation
55             dse.reset();
56             // make a unique document ID for the part
57             String partDID = dse.createResultDID();
58             // attach a document property indicating the document type
59             dse.setDocType("textline");
60             dse.setProperty("length", ""+lines[i].length());
61             dse.setProperty("content", lines[i]);
62             // register the split
63             dse.commit();
64         }
65     }
66     catch (Exception e) {
67         dse.fail("Unable to split input!", e);
68     }
```

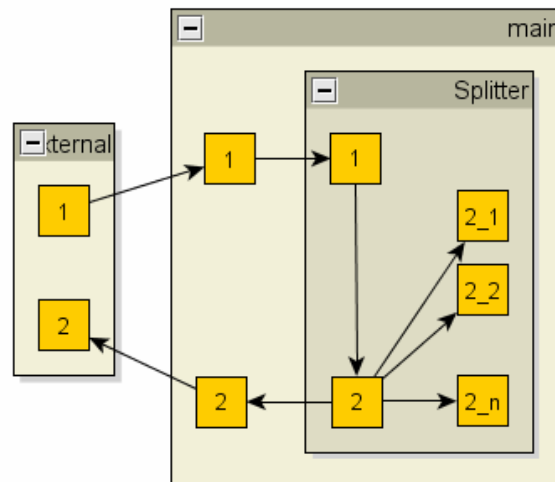


Figure 8: Splitting document into fragments

Figure 8 illustrates the splitting added by the latest example. Normally the fragments would somehow flow onwards in the system. The next example show how this might be recorded.

6.5 Example: communicating client/server applications

The next example extends the previous splitting sub-transformer such that each document fragment is sent to another separate application running in a another JVM. Communication is done using sockets.

The additional program acts as a simple server that accepts incoming translation requests. The client program (the splitting sub-transformer) sends each fragment (line from transformed document) to the server which then lowercases and reverses the line. A real server might be able to do something more useful like translate the line from one language to another.

The sub-transformer is implemented in `SplittingCommunicatingSubTransformer` and the service in `TranslatorService`.

Since the two programs are only connected through a network socket, we have to exchange both the document fragment and the corresponding Document ID. The client will do its splitting as before but also register the send of fragments and receivment of the translated fragments.

The service will register events when receiving, transforming and sending documents. The service will create a new transformer instance each time it is invoked. This is important and means that the service will have multiple sub-transformer instances under its main transformer instance. It would be perfectly possible to have one (or no) single sub-transformer in the service but we wish to illustrate the transformer instance principle here as well. In the following figure this is depicted by the three `translator` instances. The complete document exchange is depicted in Figure 9. Note that we have grouped by machine on the highest level, then distinct `jvm+application` instances and from this level on transformer hierarchy.

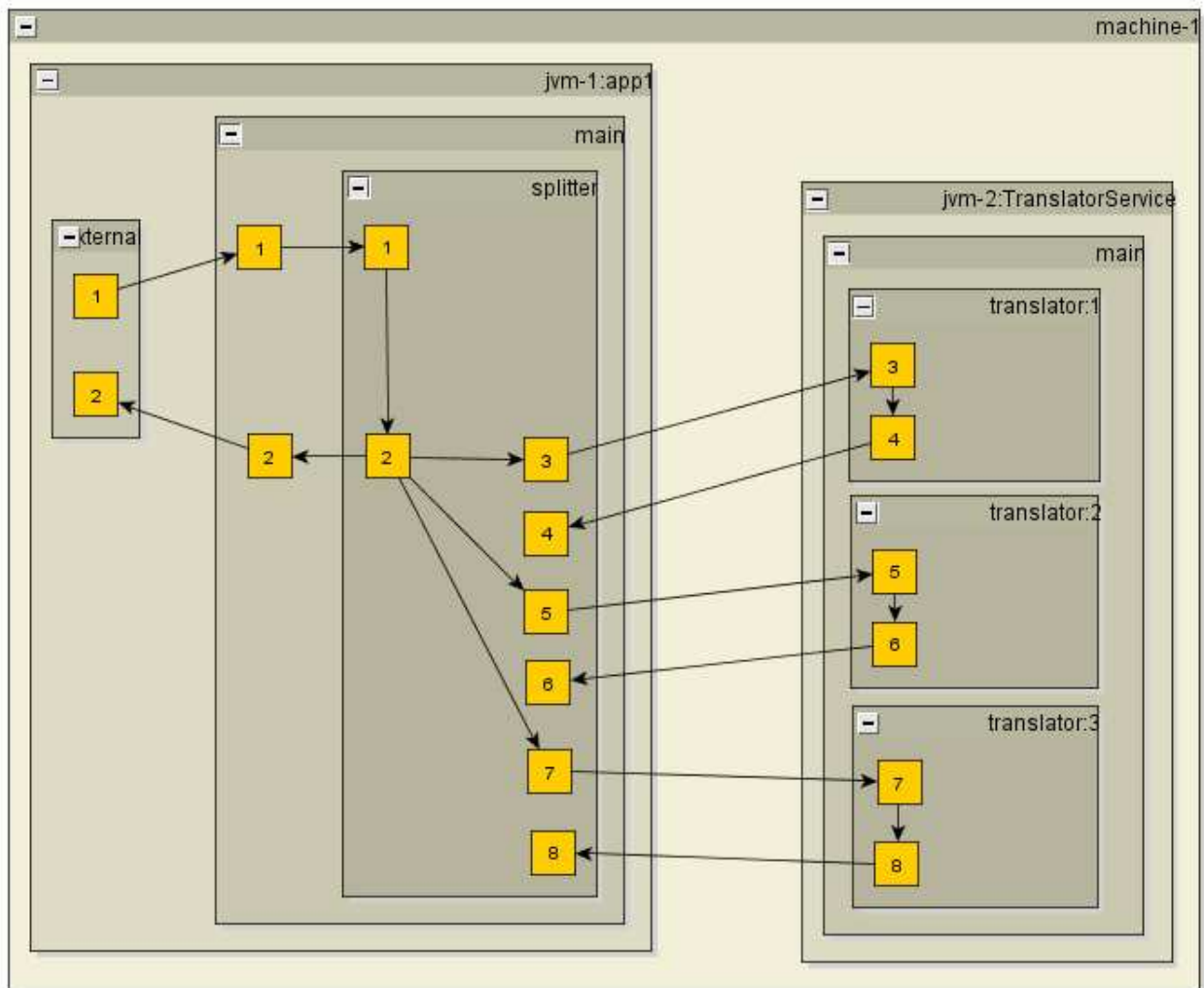


Figure 9. Two applications communicating documents between separate JVM's

Figure 9 shows the graph that could be constructed from the events generated by the two programs involved. Lets have a look at the actual code with the necessary event. Most of the code is exactly as in the previous examples.

6.5.1 TranslatorService

We first present the server code. Listing 4 shows the first section of the server. The root transformer is created (main) and a server socket is created. Next the server starts accepting incoming requests. Each received line (line 32) is perceived as a document and handled by a new sub-transformer instance (line 34). The line is formatted as `<id>#<line for translation>`. I.e. the first part up to the hash-sign is the document ID. In lines 36-38 the ID and the text content is split into two distinct variables (id and input).

```
17 public class TranslatorService {
18
19     public void runExample() throws Exception {
20         /**
21          * Obtain a root transformer instance and name it 'main'.
22          */
23         TID tid = ADKTraceFactory.createRootTID(ADKLogFactory.getAppID()
24             , "main", null);
25
26         ServerSocket s = new ServerSocket(5543);
27         while(true) {
28             Socket sock = s.accept();
29             BufferedReader br = new BufferedReader(new InputStreamReader(sock.getInputStream()));
30             BufferedWriter bw = new BufferedWriter(new OutputStreamWriter(sock.getOutputStream()));
31             String line;
32             while( (line=br.readLine())!=null) {
33                 // a new sub instance every time we get invoked (for the sake of the example)
34                 TID subTID = tid.createChildTID("translatorInstance");
35
36                 String split[] = line.split("#");
37                 String id = split[0];
38                 String input = split[1];
39             }
40         }
41     }
42 }
```

Listing 4: Transformation server receives documents and ids

In the following lines a receive event is made (line 43-44). This event uses the received document ID. Next the string “translation” and corresponding Transform event takes place (49-59). Finally a Send event is generated and hence a new document ID emerges (for the transformed document). The resulting document and new document ID are returned to the client using the same line format that was received – I.e. <id>#<text>.

```

40         /**
41         * register receivment
42         */
43         ReceiveEvent re = subTID.getReceiveEvent();
44         re.commit(id);
45         // just for debug..
46         System.out.println("input DID = '"+id+"'");
47         System.out.print("Input data = '"+input+"'");
48
49         TransformEvent te = subTID.getTransformEvent(id);
50         String resID = te.toString();
51
52         // lowercase and revers chars..
53         input = input.toLowerCase();
54         StringBuffer res = new StringBuffer();
55         for(int i=input.length()-1; i>=0; i--) {
56             res.append(input.charAt(i));
57         }
58         System.out.println(" --> '"+res+"'");
59         te.commit();
60         /**
61         * Send reply and make send event..
62         */
63         SendEvent se = subTID.getSendEvent(resID);
64         String sendID = se.toString();
65         try {
66             bw.write(sendID + "#" + res + "\n");
67             bw.flush();
68             se.commit();
69         } catch (Exception e) {
70             se.fail("Unable to send translated document", e);
71         }
72         System.out.println("-----");
73     }

```

Listing 5: generating R/T/S events in the TranslatorService

6.5.2 Client code

The client side of the document exchange is quite similar to what we have seen so far. A main transformer (main) invokes a sub-transformer (splitter) that transform (uppercases) and splits the transformed document into lines. Each line is sent to the server and a send/receive event is generated for each fragment.

We shall only list the modified sub-transformer code here since the remaining code is similar to the previous examples.

```
57         for(int i=0; i<lines.length; i++) {
58             // clear the held data in the splitter if any from
59             // previous split operation
60             dse.reset();
61             // make a unique document ID for the part
62             String partDID = dse.createResultDID();
63             // attach a document property indicating the document type
64             dse.setDocType("textline");
65             dse.setProperty("length", ""+lines[i].length());
66             dse.setProperty("content", lines[i]);
67             // register the split
68             dse.commit();
69
70             /**
71              * Invoke external application for a translation of the
72              * line
73              */
74             DPair dp = new DPair(partDID);
75             dp.setSource(partDID);
76             try {
77                 String res = externalTranslate(tid, lines[i], dp);
78                 String partResId = dp.getResult();
79                 // we dont use the id further in this example but could
80                 // have stored it in a database, merged all fragments etc.
81             } catch (Exception e) {
82
83             }
84         }
85     }
```

Listing 6. Communicating client sending document fragments

The code in Listing 6 shows how the client invokes a local method (`externalTranslate()`) for each document fragment split from the original transformed document. We use a `DPair` object so we can convey document id's in/out of the method though we don't actually use the resulting id in this example.

If we look into the `externalTranslate()` method we see the actual socket communication with the service previously described. In line 108-109 we initialize a send event for the following document sending. The actual sending of the document and the generated id through a socket channel is achieved in lines 112-122 (including send event commit/fail).

Once we have sent the data successfully we try to read back the reply. Obviously this should be accompanied with a receive event. This is done in the lines 126-139. Note that the document id of the resulting (returned) document is the one that must be set as the result in the `DPair` object (line 133).

```

101 private String externalTranslate(TID tid, String string, DPair dp) throws Exception {
102     String res = null;
103     Socket sock =null;
104     /**
105      * Initialize a new send event since we wish the remote service to receive this
106      * document.
107      */
108     SendEvent se = tid.getSendEvent(dp.getSource());
109     String sentId = se.toString();
110     BufferedReader br = null;
111     try {
112         try {
113             sock = new Socket("localhost", 5543);
114             br = new BufferedReader(new InputStreamReader(sock.getInputStream()));
115             BufferedWriter bw = new BufferedWriter(new OutputStreamWriter(sock.getOutputStream()));
116             bw.write(sentId + "#" + string + "\n");
117             bw.flush();
118             se.commit();
119         } catch (Exception e) {
120             se.fail("Unable to send document", e);
121             throw e;
122         }
123         /**
124          * Init a receive event.
125          */
126         ReceiveEvent re = tid.getReceiveEvent();
127         try {
128             String line = br.readLine();
129             String split[] = line.split("#");
130             String rcvdId = split[0];
131             res = split[1];
132             // set the received id as the resulting id
133             dp.setResult(rcvdId);
134             // commit the receive using the id from peer.
135             re.commit(rcvdId);
136         } catch (Exception e) {
137             re.fail("Error receiving response", e);
138             throw e;
139         }
140     } finally {
141         if(sock!=null) {
142             try {
143                 sock.close();
144             } catch (Exception e) { }
145         }
146     }
147     return res;
148 }

```

6.5.3 Resulting database entries

This section shows some of the trace data persisted in the database as the result of executing the client/server programs described above. Three lines of input was given to the client from stdin and a Ctrl-Z ended input.

The persistence classes have correctly assigned all traces generated to the same trace instance. This is seen from the Trace table shown in Figure 10. The trace has ID 1 and no indication of any errors.

ID	FIRSTEVENT	LASTEVENT	ERROR
1	2006-05-16 10:16:54.655	2006-05-16 10:17:01.155	0

Figure 10. Resulting trace record

If we look at the TRACEEVENT table we see all events that were generated from all transformers involved. The RUNID refers to the JVM/APP/Installation combination from which each event was received.

ID	RUNID	LOGGROUP	TRA...	TID	TIMESTAMP	EVENT	DOCUMENT
131072	65536	98304	1	1	2006-05-16 10:16:54.655	r	32768
131073	65536	98304	1	1	2006-05-16 10:16:57.421	s	32768
131074	65536	98305	1	1.1	2006-05-16 10:16:57.491	r	32768
131075	65536	98305	1	1.1	2006-05-16 10:16:57.521	t	32768
131076	65536	98305	1	1.1	2006-05-16 10:16:57.591	S	32769
131077	65536	98305	1	1.1	2006-05-16 10:16:57.651	s	32770
229376	163840	196608	1	1.1	2006-05-16 10:16:57.82	r	32770
229377	163840	196608	1	1.1	2006-05-16 10:17:00.476	t	32770
229378	163840	196608	1	1.1	2006-05-16 10:17:00.536	s	262144
131078	65536	98305	1	1.1	2006-05-16 10:17:00.596	r	262144
131079	65536	98305	1	1.1	2006-05-16 10:17:00.626	S	32769
131080	65536	98305	1	1.1	2006-05-16 10:17:00.666	s	32771
229379	163840	196608	1	1.2	2006-05-16 10:17:00.676	r	32771
229380	163840	196608	1	1.2	2006-05-16 10:17:00.736	t	32771
229381	163840	196608	1	1.2	2006-05-16 10:17:00.756	s	262145
131081	65536	98305	1	1.1	2006-05-16 10:17:00.826	r	262145
131082	65536	98305	1	1.1	2006-05-16 10:17:00.846	S	32769
229382	163840	196608	1	1.3	2006-05-16 10:17:00.906	r	32772
131083	65536	98305	1	1.1	2006-05-16 10:17:00.926	s	32772
229383	163840	196608	1	1.3	2006-05-16 10:17:00.996	t	32772
229384	163840	196608	1	1.3	2006-05-16 10:17:01.025	s	262146
131084	65536	98305	1	1.1	2006-05-16 10:17:01.055	r	262146
131085	65536	98305	1	1.1	2006-05-16 10:17:01.075	s	32769
131086	65536	98304	1	1	2006-05-16 10:17:01.125	r	32769
131087	65536	98304	1	1	2006-05-16 10:17:01.155	s	32769

Figure 11. Trace events

The LOGGROUP refers to the fully qualified name of the transformer that generated the event – the actual value is located in the LG table (Figure 12).

The TID column gives the *instance* of the transformer. E.g. since we sent three lines for transformation in the service and the service created a new sub-transformer instance every time we have the instance id's: 1.1, 1.2 and 1.3 that all are of type `main.translator`. From the instance id we can deduce that 1.1, 1.2 and 1.3 are three transformer instances (1,2 and 3) that have the parent instance 1.

LGID	LOGGROUP
98304	main
98305	main.Splitter
196608	main.translator

Figure 12. Log groups

All unique document id's are stored in the DOCUMENT table shown in Figure 13.

ID	DID	TRACEID
32768	YmJvLWxhcHRvcA==.NzUwMTU5OjEwYjNjNDQ3NTQxOi04MDAw.1	1
32769	YmJvLWxhcHRvcA==.NzUwMTU5OjEwYjNjNDQ3NTQxOi04MDAw.2	1
32770	YmJvLWxhcHRvcA==.NzUwMTU5OjEwYjNjNDQ3NTQxOi04MDAw.3	1
262144	YmJvLWxhcHRvcA==.MzkwMWM2OjEwYjNjNDRIYmU2Oi04MDAw.1	1
32771	YmJvLWxhcHRvcA==.NzUwMTU5OjEwYjNjNDQ3NTQxOi04MDAw.4	1
262145	YmJvLWxhcHRvcA==.MzkwMWM2OjEwYjNjNDRIYmU2Oi04MDAw.2	1
32772	YmJvLWxhcHRvcA==.NzUwMTU5OjEwYjNjNDQ3NTQxOi04MDAw.5	1
262146	YmJvLWxhcHRvcA==.MzkwMWM2OjEwYjNjNDRIYmU2Oi04MDAw.3	1

Figure 13. Documents

All events that are used for communicating documents (send/receive events) have an entry in the IOEVENT table. This table extends the TRACEEVENT table for s/r events and holds the additional information about the path that the send/received document traversed and optional information about the peer to/from which the document was sent/received. Note that the PEERDESCRIPTION is only to be used when documents are exchange with external parties that don't produce s/r events themselves. For example in the case where we made s/r events when writing/reading to stdout/stdin. In these cases the programs should store a description of the peer that can be used when presenting the trace later.

The path of a document is constructed by the send event. The send event encodes the path into the document id that is used programmatically through the API. Whenever a send generates a document ID it adds a path-suffix to the existing path of the document. I.e. if a document is received with a path component equal to 2.1.4.2, then the send event will make a path like 2.1.4.2.1 and encoded it together with the document id. Since this id is sent to the receiving peer, the receiving peer will register the receipt of the document *and* the path. *This is what enable us to correlate sent and received document instances across applications and machines!* It also enable us to send the same document multiple times to different peers and still keep track of what sends went where.

When presenting the document exchange this information becomes important since a send of document D1 with path 1.2.3 from transformer `main.sub` (instance 1.1) and a receive of document D1 with path 1.2.3 in transformer `trans1.sub1.sub2` (instance 3.2.1) must be show as connected (see Figure 14).

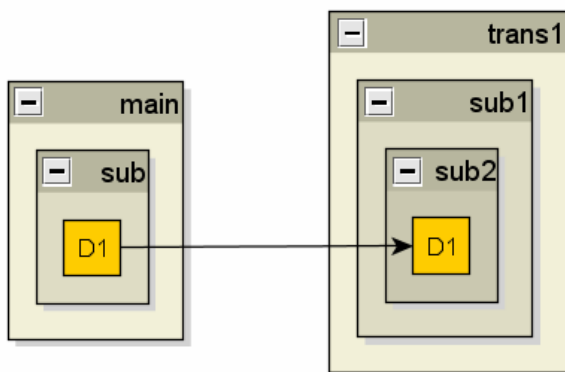


Figure 14. Visualization of document, transformers and paths

ID	PATH	PEERDESCRIPTION
131072		stdin
131073	1	<null>
131074	1	<null>
229376	2	<null>
229378	1	<null>
131077	2	<null>
131078	1	<null>
229379	3	<null>
229381	2	<null>
131080	3	<null>
131081	2	<null>
229382	4	<null>
229384	3	<null>
131083	4	<null>
131084	3	<null>
131085	5	<null>
131086	5	<null>
131087	5.6	stdout

Listing 7. IOEvents

7 Applying events to a Custom AXT Filter

An AXT filter is typically part of a larger transformation. All AXT Filters inherit from the AbstractAXTFilter super class which implements the default events for a filter. However when writing a custom filter there might be situations where it would be convenient to directly access the Transformer or the document id, for instance if the filter should generate checkpoint events.

The methods used in the AXTAbstract filter are therefore publicly available for all custom filters to use and are accessed using the following getter and setter methods:

getTID()	Returns the transformations id (TID) for the current transformation
getDPair()	Returns a pair of document ids (DID), since a filter typically changes data and thus results in a new document filter operates with a source document and a result document, these may be the same, but are typically not.
getDPair().getSource()	The document id (DID) which is the input to the filter.
getDPair().getResult()	The document id (DID) which is the output from the filter.

8 References

- [1] http://en.wikipedia.org/wiki/Graph_theory
- [2] <http://en.wikipedia.org/wiki/Base64>