How Jonathan propagates a transactional context

Kathleen Milsted
kathleen.milsted@francetelecom.fr

May 2000

The aim of this article is to describe the main interfaces and classes used by Jonathan to transmit a transactional context with messages. The implementation is closely inspired by the relevant OMG specifications (see references), with some additions where the specs are incomplete on certain issues.

The interfaces and classes described here pertain to Jonathan release 2.0.

1 ORB Service Contexts

A service context contains information specific to an ORB service (e.g., a transaction service) for an individual request or reply message. Request messages are initiated by clients, while reply messages are initiated by servers. A service context consists of an ID (an unsigned long) identifying the service, plus a sequence of bytes encoding the actual context data.

At the date of writing, the following service IDs have been attributed by the OMG[omg:interop, section 13.6.7]:

0	=	Transaction service
1	=	Codeset negotiation service
2,3,4	=	DCOM-CORBA interworking service
5	=	Bi-directional IIOP service
6	=	Object by value service
7	=	Asynchronous messaging
8	=	Firewall
9	=	Java throwable service

Note that each service requiring context information to be passed by the GIOP protocol [omg:giop] must be assigned a unique ID by the OMG. Service ID 0 has been assigned since GIOP 1.0, service ID 1 since GIOP 1.1 and the other IDs since the current version GIOP 1.2.

Service contexts are transmitted in the headers of GIOP request and reply messages. The header of these messages thus contains a (possibly empty) list of service contexts.

In Jonathan, a service context is implemented by the following class:

```
final class org.omg.IOP.ServiceContext {
    int context_id;
    byte[] context_data;
}
```

2 The Propagation Context

The data part of a Transaction service context is referred to as a *Propagation context*. The official IDL definition of this context can be quite rich:

```
struct org.omg.CosTransactions.PropagationContext {
    unsigned long timeout;
    TransIdentity current;
    sequence <TransIdentity> parents;
    any implementation-specific-data;
}
```

However, in Jonathan+JOnAS it was decided to have a simpler definition (at least to start with), so a Propagation context is currently defined as an empty (but Serializable) interface:

```
interface org.omg.CosTransactions.PropagationContext
    extends java.io.Serializable {}
```

This interface is implemented in JOnAS by the class `org.objectweb.jonas.jtm.PropagationContext`.

Note that an `org.omg.CosTransactions.PropagationContext` is not a `org.omg.IOP.ServiceContext`; the former must be encoded into the data attribute of the latter.

3 Interface between the ORB and a Transaction service

The CORBA Transaction service spec defines two elements of the interface between the ORB and a Transaction service, one on the ORB side, the other on the Transaction service side:

- on the ORB side, an interface that the Transaction service (when present) invokes to identify itself to the ORB. We refer to the object implementing this interface as the ORB's *Transaction Service handler*. (TSHandler). However, note that the OMG spec does not use this terminology.
- on the Transaction service side, an interface that the ORB invokes to poll the Transaction service when a request or reply is sent or received. This interface is composed of two parts, a *Sender* and a *Receiver*.

In addition, Jonathan introduces a further element:

- on the ORB side, an interface that is invoked (internally) by the ORB to poll all identified services when a request or reply is sent or received. We refer to this interface as the ORB's *Services Handler*.

3.1 The ORB Transaction Service Handler

This is an object that the ORB provides. According to [omg:ts], the ORB's TSHandler (but remember that this is not its name in the spec) must implement the following interface `org.omg.CORBA.TSIdentification` to allow a Transaction service to identify itself to the ORB:

```
interface org.omg.CORBA.TSIdentification {
    void identify_sender(org.omg.CosTSPortability.Sender s)
         throws NotAvailable, AlreadyIdentified;
    void identify_receiver(org.omg.CosTSPortability.Receiver r)
         throws NotAvailable, AlreadyIdentified;
}
```

Thus, identifying a Transaction service to an ORB means identifying the Sender and Receiver (see later) of the Transaction service. The above interface is implemented in Jonathan by the class `org.objectweb.david.libs.services.CosTransactions.handler.TSHandler`. Since it is defined in the David (CORBA) personality of Jonathan, we call this the David Transaction Service handler.

As well as allowing a Transaction service to identify itself, the David TSHandler implements another interface as follows:

```
interface org.objectweb.david.apis.services.handler.Service {

    org.omg.IOP.ServiceContext
                    getRequestContext(int request_id,
                                      boolean response_expected,
                                      byte[] object_key);

    org.omg.IOP.ServiceContext getReplyContext(int request_id);

    void handleRequestContext(org.omg.IOP.ServiceContext context,
                              int request_id,
                              boolean response_expected,
                              byte[] object_key);

    void handleReplyContext(org.omg.IOP.ServiceContext context,
                            int request_id);
}
```

This interface (which is specific to Jonathan and which is not defined in the OMG spec) is the means by which the ORB interacts (indirectly) with the Transaction service's sender and receiver. For example, when a request message is being prepared for transmission client side, the ORB invokes the TSHandler's `getRequestContext` method, and the TSHandler in turn invokes the sender's `sending_request` method (see below). Likewise, when a request message is received server side, the ORB invokes the TSHandler's `handleRequestContext` method, which in turn invokes the receiver's `received_request` method.

To complicate things a little however, the David TSHandler is an abstract class even though it does indeed provide an implementation of the two

interfaces above. What makes it abstract is the way to encode a PropagationContext into the data part (an array of bytes) of a ServiceContext. This functionality is provided concretely by the Jeremie TSHandler `org.objectweb.jeremie.libs.services.CosTransactions.handler.JRMITSHandler`. which extends the David TSHandler with two methods as follows:

```
org.omg.IOP.ServiceContext
   encodeContext(org.omg.CosTransactions.PropagationContext ctx);
org.omg.CosTransactions.PropagationContext
   decodeContext(org.omg.IOP.ServiceContext sc);
```

The implementation of these methods by the Jeremie TSHandler makes use of standard Java serialization (java.io.ObjectOutputStream, java.io.ObjectInputStream) so can not be included in David, which is meant to be a pure CORBA personality.

3.2 The Transaction Service Sender

This is an object that the Transaction service provides. It has the following interface:

```
interface org.omg.CosTSPortability.Sender {

   org.omg.CosTransactions.PropagationContext
       sending_request (int reqId);

   void received_reply (int reqId,
                        org.omg.CosTransactions.PropagationContext ctx,
                        org.omg.CORBA.Environment env);
}
```

The sender is called by the TSHandler on the client side whenever:
- a request message is sent (`sending_request`)
- a corresponding reply message is received (`received_reply`)

This interface is implemented in JOnAS by the class `org.objectweb.jonas.rmifilters.JonasSender`.

3.2.1 The Transaction Service Receiver

This is an object that the Transaction service provides. It has the following interface:

```
interface org.omg.CosTSPortability.Receiver {

   org.omg.CosTransactions.PropagationContext
       sending_reply (int reqId);

   void received_request(int reqId,
                         org.omg.CosTransactions.PropagationContext ctx);
}
```

The receiver is called by the TSHandler on the server side whenever:
- a request message is received (`received_request`)
- a corresponding reply message is sent (`sending_reply`)

This interface is implemented in JOnAS by the class `org.objectweb.jonas.rmifilters.JonasReceiver`.

3.3 The ORB Services Handler

This is an object that the ORB provides. It is specific to Jonathan and is not defined in the OMG specs. It is called internally in the ORB by the GIOP protocol to poll all identified services when a request or reply message is sent or received. The interface defining this object is a "plural" version of the interface `org.objectweb.david.apis.services.handler.Service` above, which is used when only a Transaction service is polled.

```
interface org.objectweb.david.apis.services.handler.ServicesHandler {

    org.omg.IOP.ServiceContext[]
                   getRequestContexts(int request_id,
                                      boolean response_expected,
                                      byte[] object_key);

    org.omg.IOP.ServiceContext[] getReplyContexts(int request_id);

    void handleRequestContexts(org.omg.IOP.ServiceContext[] context,
```

```
                                int request_id,
                                boolean response_expected,
                                byte[] object_key);

    void handleReplyContexts(org.omg.IOP.ServiceContext[] context,
                             int request_id);
}
```

This interface is implemented in Jonathan by the class `org.objectweb.david.libs.services.handler.DavidServicesHandler`.

4 The role of the Jonathan kernel

Some of the above-mentioned interfaces and classes make use of the Jonathan kernel to dynamically find and initialize other objects that they need. What follows is a brief description and illustration of this process.

Basically, a Jonathan kernel has a bind method with the following signature:

```
public Object bind(Object spec) throws JonathanException
```

The input argument is expected to be a String. When bind is invoked, the kernel first attempts to find a Java property having the string as its name. If so, the kernel assumes that the value of the property is the name of a class, which it attempts to load and then initialize. This notion of initialization means invoking a static method `init` on the class. There are no constraints on the implementation of `init`, it can do whatever it wants, including typically creating an instance (or instances) of the class being initialized.

For example, the ORB ServicesHandler, which is implemented by the class `org.objectweb.david.libs.services.handler.DavidServicesHandler`, needs to discover which ORB services are present. It therefore invokes the bind method of the kernel to "bind" any classes defined by a property of the form `david.DavidServicesHandler.`n where n is intended to be an assigned OMG service id. Thus, in the case of Jonathan+JOnAS, the following property is needed:

```
david.DavidServicesHandler.0 =
    org.objectweb.jeremie.libs.services.CosTransactions.handler.JRMITSHandler
```

Likewise, the Jeremie TSHandler discovers the JOnAS sender and receiver classes through the following two properties:

```
jeremie.JRMITSHandler.sender =
    org.objectweb.jonas.rmifilters.JonasSender

jeremie.JRMITSHandler.receiver =
    org.objectweb.jonas.rmifilters.JonasReceiver
```

For information, note that the Jonathan kernel discovers properties in the following order (descending priority):

- run-time properties specified when java is run with the -D option;
- properties specified when the kernel itself is initialized (using the static `init` method of the class `org.objectweb.jonathan.apis.kernel.Kernel`)
- custom properties of the current installation of Jonathan defined in a file named *jonathan.prop*; this file must be somewhere in the CLASSPATH if it is to be taken into account.
- default properties of the current build of Jonathan defined in a source file *org.objectweb.jonathan.apis.kernel.DefaultProperties.prop*; this file is compiled when Jonathan is built.

References

[omg:interop] ORB Interoperability Architecture. CORBA/IIOP 2.3.1 Specification, chapter 13. CORBA V2.3, June 1999.

[omg:giop] General Inter-ORB Protocol. CORBA/IIOP 2.3.1 Specification, chapter 15. CORBA V2.3.1, October 1999.

[omg:ts] Transaction Service Specification. CORBAservices Specification, chapter 10.