RESTful Services and Distributed Transactions

An example using XTA: XA Transaction API

Christian Ferrari, tiian@users.sourceforge.net, 2019

Abstract

The two-phase commit protocol was designed in the epoch of the "big iron" systems like mainframes and UNIX servers; the XA specification was defined in 1991 when the typical deployment model consisted of all the software installed in a single server. Surprisingly enough, a consistent part of the specification can be seen from a different perspective and it can be re-used to support distributed transactions inside a micro services based architecture. This post explains how LIXA and XTA enable the development of polyglot distributed transactional systems.

Introduction

In brief, the two-phase commit protocol is a specialized consensus protocol that requires two phases. The first phase, the *voting phase*, is where all the concerned resources are asked to "*prepare*": a positive reply implies that a "*durable*" state has been reached. The second phase, the *commit phase*, confirms the new state of all the concerned resources. In the event of an error, the protocol roll backs and all the resources reach the previous state.

The XA specification

One of the most common implementations of the two-phase commit protocol is the XA specification [1]: it is been supported by many middlewares developed during the Nineties and it's leveraged by the JTA specification [2].

Historical Acceptance

The usage of the two-phase commit protocol has been debated a lot since its inception. On one side, the enthusiasts tried to use it in every circumstance; on the other side, the detractors avoided it in all the situations.

A first note that must be reported is related to performance: as every consensus protocol, two-phase commit increases the time spent by a transaction. This side effect can't be avoided and it must be considered at the design time.

It's even common knowledge that some resource managers are affected by scalability limits when they manage XA transactions: this behavior depends more on the quality of the implementation than on the two-phase commit protocol itself.

The author thinks that the abuse of two-phase commit severely hurts the performance of a distributed system, but trying to avoid it when it's the obvious solution, leads to baroque and over engineered systems that are difficult to maintain. More specifically, the integration of already existing services can conduct to serious re-engineering when the transactional behavior must be guaranteed.

Architecture

LIXA is a free and open source software licensed under the terms of the GNU Public License and the Lesser GNU Public License, it can be freely downloaded from GitHub and from SourceForge [3].

XTA stands for XA Transaction API and it's an interface designed to support the contemporary programming needs: it reuses the core components developed by the LIXA project and it introduces a new programming model (see below). XTA is currently available for C, C++, Java and Python; it can be compiled from source code and installed in a Linux system or it can be executed as a Docker container.

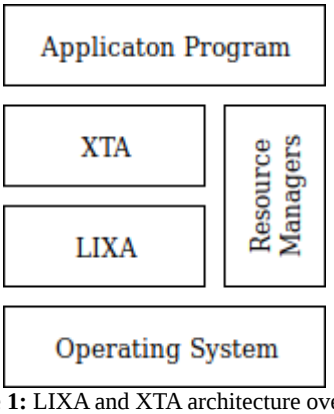

Figure 1: LIXA and XTA architecture overview

Figure 1 shows the main picture about XTA architecture: independently from the programming language, the "Application Program" interacts directly with XTA to manage transactions and with one or more "Resource Managers" to manage persistent data. Typical resource managers are MySQL/MariaDB and PostgreSQL.

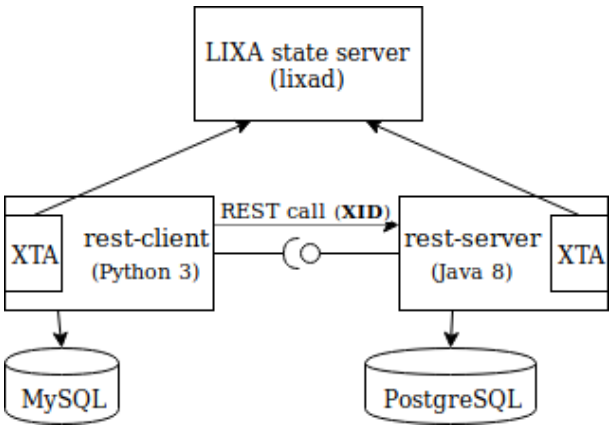

Figure 2: a transaction that spans two applications and two resource managers

Figure 2 shows the high level architecture implemented by the example explained below. More specifically, in our example:

- "rest-client" is a client program developed using Python 3; it persists data in a MySQL database
- "rest-server" is a service implemented in Java and it's called by mean of a REST API; it persists data in a PostgreSQL database
- "lixad" is the LIXA state server, it persists the transactions state on behalf of the XTA processes.

All the components are executed as Docker containers [4] for the sake of easiness, but a traditional installation can be done as well.

Executing the example

Before starting, you need an operating system with two fundamental tools: git and Docker. Both are available for free and are easy to set-up in Linux, MacOS and Windows.

Set-up

First of all, clone the git repository that contains all you need:

```
$ git clone https://github.com/tiian/lixa-docker.git
```

Then go to the example directory:

```
$ cd lixa-docker/examples/PythonJavaREST
```

Build the Docker images for the "rest-client" and for the "rest-server":

```
$ docker build -f Dockerfile-client -t rest-client .
$ docker build -f Dockerfile-server -t rest-server .
```

Check the built images, you should see something like:

```
$ docker images | grep rest
rest-server          latest              81eda2af0fd4        25 hours ago
731MB
rest-client          latest              322a3a26e040        25 hours ago
390MB
```

Start MySQL, PostgreSQL and the LIXA state server (lixad):

```
$ docker run --rm -e MYSQL_ROOT_PASSWORD=mysecretpw -p 3306:3306 -d lixa/mysql
$ docker run --rm -e POSTGRES_PASSWORD=lixa -p 5432:5432 -d lixa/postgres -c
'max_prepared_transactions=10'
$ docker run --rm -p 2345:2345 -d lixa/lixad
```

Check the started containers, you should see something like:

```
$ docker ps | grep lixa
16099992bd82        lixa/lixad          "/home/lixa/lixad-en…"   6 seconds ago
Up 3 seconds        0.0.0.0:2345->2345/tcp              sharp_yalow
15297ed6ebb1        lixa/postgres       "docker-entrypoint.s…"   13 seconds ago
Up 9 seconds        0.0.0.0:5432->5432/tcp              unruffled_brahmagupta
3275a2738237        lixa/mysql          "docker-entrypoint.s…"   21 seconds ago
Up 18 seconds       0.0.0.0:3306->3306/tcp, 33060/tcp   sharp_wilson
```

Start the programs

Activate the Java service (replace the IP address “192.168.123.35” with the IP address of your Docker host):

```
$ docker run -ti --rm -e MAVEN_OPTS="-Djava.library.path=/opt/lixa/lib" -e
LIXA_STATE_SERVERS="tcp://192.168.123.35:2345/default" -e
PQSERVER="192.168.123.35" -p 18080:8080 rest-server
```

wait for the service readiness, check the following messages in console:

```
Mar 24, 2019 9:21:45 PM org.glassfish.grizzly.http.server.NetworkListener start
INFO: Started listener bound to [0.0.0.0:8080]
Mar 24, 2019 9:21:45 PM org.glassfish.grizzly.http.server.HttpServer start
INFO: [HttpServer] Started.
Jersey app started with WADL available at http://0.0.0.0:8080/xta/applica-
tion.wadl
Hit enter to stop it...
```

Start the Python client from another terminal (replace the IP address “192.168.123.35” with the IP address of your Docker host):

```
docker run -ti --rm -e SERVER="192.168.123.35" -e LIXA_STATE_SERVERS="tcp://
192.168.123.35:2345/default" rest-client
```

At this point, you should have some messages in the console of the **Java service**:

```
01 ***** REST service called: xid='1279875137.c5b6d8bf7d584065b4ee29d62fe35ab5.ac3d62e-
b862b49fe6a50bfee46d142b9', oper='delete' *****
02 2019-03-24 21:23:04.047857 [1/139693866854144] INFO: LXC000I this process is starting a new LIXA
transaction manager (lixa package version is 1.7.6)
03 Created a subordinate branch with XID '1279875137.c5b6d8bf7d584065b4ee29d62fe35ab5.ac3d62e-
b862b49fe9de52bd41657494a'
04 PostgreSQL: executing SQL statement >DELETE FROM authors WHERE id=1804<
05 Executing first phase of commit (prepare)
06 Returning 'PREPARED' to the client
07 Executing second phase of commit

08 ***** REST service called: xid='1279875137.91f1712af1164dd38dcc0b14d58819d2.ac3d62e-
b862b49fe6a50bfee46d142b9', oper='insert' *****
09 Created a subordinate branch with XID '1279875137.91f1712af1164dd38dcc0b14d58819d2.ac3d62e-
b862b49fe20cca6a7fc7c4802'
10 PostgreSQL: executing SQL statement >INSERT INTO authors VALUES(1804, 'Hawthorne', 'Nathaniel')<
11 Executing first phase of commit (prepare)
12 Returning 'PREPARED' to the client
13 Executing second phase of commit
```

and some other messages in the console of the **Python client**:

```
01 2019-03-24 21:23:03.909808 [1/140397122036736] INFO: LXC000I this process is starting a new LIXA
transaction manager (lixa package version is 1.7.6)
02 ***** REST client *****
03 MySQL: executing SQL statement >DELETE FROM authors WHERE id=1840<
04 Calling REST service passing: xid='1279875137.c5b6d8bf7d584065b4ee29d62fe35ab5.ac3d62e-
b862b49fe6a50bfee46d142b9', oper='delete'
05 Server replied >PREPARED<
06 Executing transaction commit
07 ***** REST client *****
08 MySQL: executing SQL statement >INSERT INTO authors VALUES(1840, 'Zola', 'Emile')<
09 Calling REST service passing: xid='1279875137.91f1712af1164dd38dcc0b14d58819d2.ac3d62e-
b862b49fe6a50bfee46d142b9', oper='insert'
10 Server replied >PREPARED<
11 Executing transaction commit
```

Execution explained

Python client:

- row 01: the client starts its transaction manager
- row 03: the client executes the SQL statement (“DELETE”) in MySQL

- row 04: the client calls the Java service passing the transaction identifier (xid) and the required operation (“delete”)

Java service:

- row 01: the service is called, it receives the transaction identifier (xid) and the required operation (“delete”)
- row 02: the service starts its transaction manager
- row 03: the service branches the global transaction and it creates a new transaction identifiers
- row 04: the service executes the SQL statement (“DELETE”) in PostgreSQL
- row 05: the service executes the first phase of the commit protocol, it “prepares” PostgreSQL
- row 06: the service returns the result “PREPARED” to the client
- row 07: the service performs the second phase of the commit protocol

Python client:

- row 05: the client receives the result “PREPARED” from the service
- row 06: the client executes the commit protocol

The remaining steps repeats the same actions for the second SQL statement (“INSERT”).

XTA programming model

The client/server example described above implements one of the patterns supported by XTA: “multiple applications, concurrent branches/pseudo synchronous”.

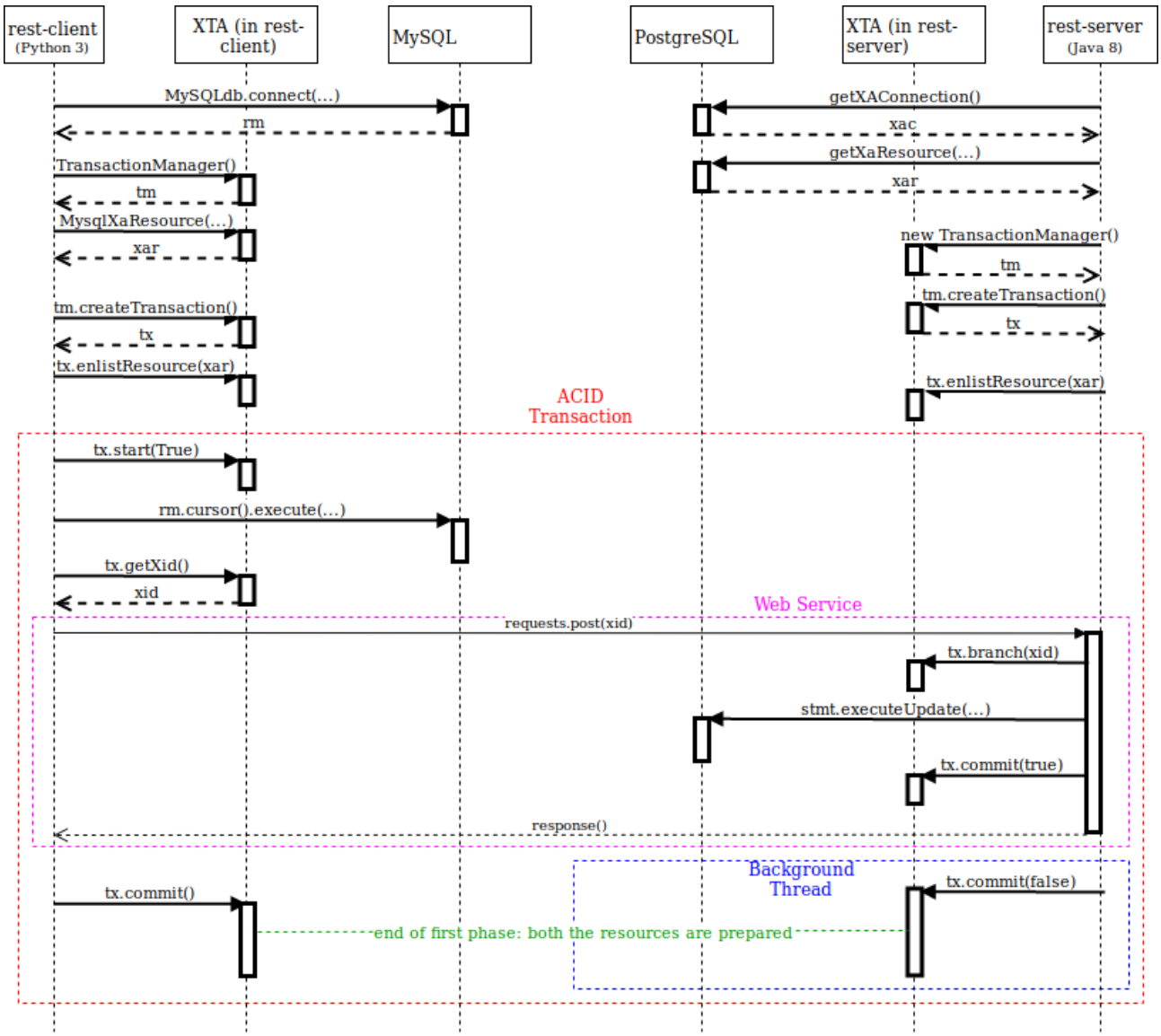

Figure 3: the “multiple applications, concurrent branches/pseudo synchronous” pattern

The diagram in Figure 3 describes the interactions between the client and the server.
The transactional part of the sequence diagram is delimited by a red dashed box.
The magenta dashed box contains the REST call and the first phase of the commit protocol: `tx.commit()` is called passing `true` as the value for “non blocking” parameter.
The blue dashed box contains the second phase of the commit protocol.
Figure 3 does not show the interactions among “rest-client”, “rest-server” and the LIXA state server:

- the magic happens when the background thread of the Java server calls `tx.commit(false)`
- the LIXA state server recognizes the multiple branches transaction and it blocks “rest-server” until “rest-client” completes its first phase of the commit
- at that point, marked by a dashed green line in the picture, a global consensus has been reached
- finally, both the players can go on with the second phase of the commit protocol.

The below pieces of code are provided to show how XTA objects and methods must be used.

The Python client code

Ignoring boilerplate and scaffolding, here is the interesting part of the Python client source code:

```
    44 # initialize XTA environment
    45 Xta_init()

    46 # create a new MySQL connection
    47 # Note: using MySQLdb functions
    48 rm = MySQLdb.connect(host=hostname, user="lixa", password="passw0rd", db="lixa")

    49 # create a new XTA Transaction Manager object
    50 tm = TransactionManager()

    51 # create an XA resource for MySQL
    52 # second parameter "MySQL" is descriptive
    53 # third parameter "localhost,0,lixa,,lixa" identifies the specific database
    54 xar = MysqlXaResource(rm._get_native_connection(), "MySQL",
    55         hostname + "/lixa")

    56 # Create a new XA global transaction and retrieve a reference from
    57 # the TransactionManager object
    58 tx = tm.createTransaction()

    59 # Enlist MySQL resource to transaction
    60 tx.enlistResource(xar)

    61 sys.stdout.write("***** REST client *****\n")
    62 # Start a new XA global transaction with multiple branches
    63 tx.start(True)

    64 # Execute DELETE statement
    65 sys.stdout.write("MySQL: executing SQL statement >" + delete_stmt + "<\n")
    66 cur = rm.cursor()
    67 cur.execute(delete_stmt)

    68 # Retrieving xid
    69 xid = tx.getXid().toString()

    70 # Calling server passing xid
    71 sys.stdout.write("Calling REST service passing: xid='" + xid + "', oper='delete'\
n")
    72 r = requests.post("http://" + hostname + ":18080/xta/myresource",
    73         data={'xid':xid, 'oper':'delete'})
    74 sys.stdout.write("Server replied >" + r.text + "<\n")

    75 # Commit the transaction
    76 sys.stdout.write("Executing transaction commit\n")
    77 tx.commit()
```

- row 54: the XA resource (xar) is linked to a MySQL connection (rm)
- row 60: the XA resource is enlisted to a transaction object

- row 63: the global transaction is started
- row 72: the client calls the REST service
- row 77: the transaction is committed

The full source code is available here:
https://github.com/tiian/lixa-docker/blob/master/examples/PythonJavaREST/client.py

The Java server code

Ignoring boilerplate and scaffolding, the example uses the Jersey framework, here is the interesting part of the Java server source code:

```
71            // 1. create an XA Data Source
72            xads = new PGXADataSource();
73            // 2. set connection parameters (one property at a time)
74            xads.setServerName(System.getenv("PQSERVER"));
75            xads.setDatabaseName("lixa");
76            xads.setUser("lixa");
77            xads.setPassword("passw0rd");
78            // 3. get an XA Connection from the XA Data Source
79            xac = xads.getXAConnection();
80            // 4. get an XA Resource from the XA Connection
81            xar = xac.getXAResource();
82            // 5. get an SQL Connection from the XA Connection
83            conn = xac.getConnection();
84            //
85            // XTA code
86            //
87            // Create a mew XTA Transaction Manager
88            tm = new TransactionManager();
89            // Create a new XA global transaction using the Transaction
90            // Manager as a factory
91            tx = tm.createTransaction();
92            // Enlist PostgreSQL resource to transaction
93            tx.enlistResource(xar, "PostgreSQL",
94                              System.getenv("PQSERVER") + ";u=lixa;db=lixa");
95            // create a new branch in the same global transaction
96            tx.branch(xid);
97            System.out.println("Created a subordinate branch " +
98                               "with XID '" + tx.getXid().toString() + "'");
99            //
100           // Create and Execute a JDBC statement for PostgreSQL
101           //
102           System.out.println("PostgreSQL: executing SQL statement >" +
103                              sqlStatement + "<");
104           // create a Statement object
105           stmt = conn.createStatement();
106           // Execute the statement
107           stmt.executeUpdate(sqlStatement);
108           // close the statement
109           stmt.close();
110           // perform first phase of commit (PREPARE ONLY)
111           System.out.println("Executing first phase of commit (prepare)");
112           tx.commit(true);
113           // start a backgroud thread: control must be returned to the client
114           // but finalization must go on in parallel
115           new Thread(new Runnable() {
116                 @Override
117                 public void run() {
118                    try {
119                        // perform second phase of commit
120                        System.out.println("Executing second phase of commit");
121                        tx.commit(false);
122                        // Close Statement, SQL Connection and XA
123                        // Connection for PostgreSQL
124                        stmt.close();
125                        conn.close();
126                        xac.close();
127                    } catch  (XtaException e) {
128                        System.err.println("XtaException: LIXA ReturnCode=" +
```

```
129                                     e.getReturnCode() + " ('" +
130                                     e.getMessage() + "')");
131                         e.printStackTrace();
132                     } catch (Exception e) {
133                         e.printStackTrace();
134                     }
135                 }
136             }).start();
137         } catch (XtaException e) {
138             System.err.println("XtaException: LIXA ReturnCode=" +
139                             e.getReturnCode() + " ('" +
140                             e.getMessage() + "')");
141             e.printStackTrace();
142         } catch (Exception e) {
143             e.printStackTrace();
144         }
145         System.out.println("Returning 'PREPARED' to the client");
146         return "PREPARED";
```

- row 81: the XA resource (xar) is retrieved by the PostgreSQL connection (xac)
- row 93: the XA resource is enlisted to a transaction object
- row 96: a new branch of the global transaction is created
- row 112: the first phase of the commit protocol (“prepare”) is executed
- row 121: the second phase of the commit protocol is executed by a background thread
- row 146: the service returns the result to the caller

The full source code is available here:
https://github.com/tiian/lixa-docker/blob/master/examples/PythonJavaREST/src/main/java/org/tiian/lixa/xta/examples/MyResource.java

XTA documentation

The full documentation of XTA is available inside the LIXA manual, XTA API reference is available for the C, C++/Python and Java languages.

LIXA and XTA uniqueness

LIXA has been designed and developed with a clear and well defined architecture: all the transactional information is persisted by the *state server* (**lixad**), most of the transactional logic is managed by the *client library* (**lixac** and its derivatives). The strong decoupling between the “logic” and the “state” enables the embedding of the transaction management capabilities in the “Application Program” without requiring neither frameworks nor application servers.
XTA pushed the concept of “distributed transaction” a step further: client and server don’t have to be executed by any sort of “supervisor middleware” because they coordinate themselves autonomously interacting with the LIXA *state server*. The only information that the client must pass to the server is the ASCII string representation of the XID. Other approaches implemented in the past required configuration and coordination among application servers; most of the times, even the communication protocols had to be aware about the transactionality aspects.
Furthermore, XTA supports multiple client/servers in the same global transaction: the same XID can be branched by many called services, even hierarchically.
XTA supports synchronous protocols, like the RESTful one shown in this example, as well as asynchronous protocols by mean of a different pattern ("multiple applications, concurrent branches/ pseudo asynchronous").

Conclusions

The XTA API in conjunction with the LIXA state server enables the development of systems that implement ACID [5] distributed transactions among 2 or more applications (services). XTA enables the development of polyglot transactional systems that use multiple resource managers and any communication protocol without requiring some sort of application manager.

References

[1] Distributed Transaction Processing: The XA Specification, X/Open:
https://publications.opengroup.org/c193
[2] Java Transaction API (JTA):
https://download.oracle.com/otn-pub/jcp/jta-1_2-mrel2-eval-spec/JTA1.2Specification.pdf
[3] LIXA project: http://www.tiian.org/lixa/

[4] LIXA Docker images are available in Docker-Hub: https://hub.docker.com/search?q=lixa&type=image
[5] ACID: Atomicity, Consistency, Isolation, Durability: https://en.wikipedia.org/wiki/ACID_(computer_science)