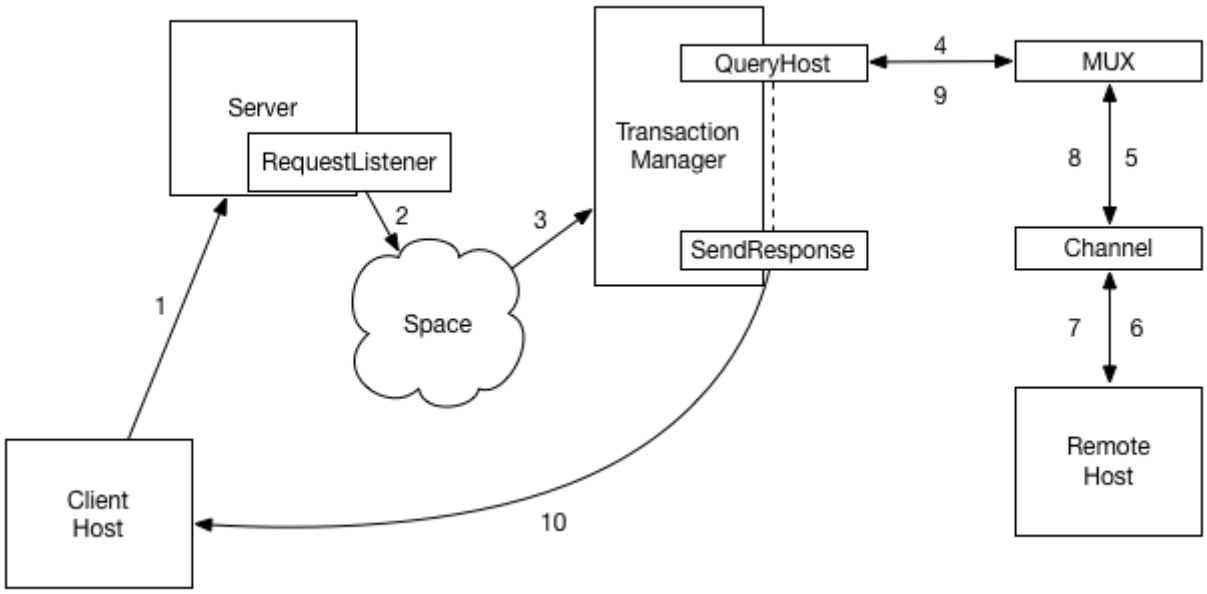
Implementing a jPOS Gateway

Table of Contents

This tutorial will show you how to configure a jPOS gateway.

First part will show you how to get it running. Second one will explain you how and why it works.

By the end of this short tutorial, you'll have a working server capable of processing a significant number of sustained TPS (Transactions Per Second), way over 1000 when running in a fast network, using the TransactionManager to send messages through a MUX to a Channel connected to a remote host.



==

How to get help

If you have questions while trying this tutorial, feel free to contact support@jpos.org, we'll be happy to help.

If you want online assistance, you can join the jPOS Slack, please request an invite.

PART 1: Get it running

Download a binary distribution of jPOS

Download http://jpos.org/downloads/jpos-2.1.1-SNAPSHOT.tar.gz or http://jpos.org/downloads/jpos-2.1.1-SNAPSHOT.zip and extract it in a working directory, for example `/opt/local`.

TIP

We typically extract in `/opt/local` and then `symlink` it to `jpos`, i.e.:

```
ln -s /opt/local/jpos-{jpos-version}-SNAPSHOT /opt/local/jpos
```

You can validate the download hashes here:

- http://jpos.org/downloads/jpos-2.1.1-SNAPSHOT.tar.gz.sha1
- http://jpos.org/downloads/jpos-2.1.1-SNAPSHOT.zip.sha1

Test run

Run

```
/opt/local/jpos/bin/q2
```

and you'd see something like this:

```
<log realm="Q2.system" at="2017-05-06T20:58:57.419">
  <info>
    Q2 started, deployDir=/opt/local/jpos/deploy
  </info>
</log>
...
...
```

Keep the server running in one terminal, then open another terminal in order to continue with the next step.

Configure a server

We'll configure a QServer listening to port 8000 using `XMLPackager` and `XMLChannel` so that we can easily use `telnet` or `netcat` to fire XML-formatted messages.

Go to the `/opt/local/jpos/deploy` directory and add a file called `50_xml_server.xml` with the following content:

```
<server class="org.jpos.q2.iso.QServer" logger="Q2"
        name="xml-server-8000" realm="xml-server-8000">
    <attr name="port" type="java.lang.Integer">8000</attr>                  ①
    <channel class="org.jpos.iso.channel.XMLChannel"                        ②
             packager="org.jpos.iso.packager.XMLPackager"                   ③
             type="server"
             logger="Q2"
             realm="xml-server-8000">
        <property name="timeout" value="180000"/>
    </channel>
    <request-listener class="org.jpos.iso.IncomingListener" logger="Q2"
                      realm="incoming-request-listener">
        <property name="queue"  value="TXNMGR" />                           ④
        <property name="ctx.DESTINATION"  value="jPOS-AUTORESPONDER" />     ⑤
    </request-listener>
</server>
```

① We listen to port 8000

② Using XMLChannel

③ And XMLPackager

④ We queue incoming messages through a Space queue arbitrarily named `TXNMGR`

⑤ Configure `IncomingListener` to place an arbitrary variable in the TM's Context named `DESTINATION`

(you can download it from http://jpos.org/downloads/tutorials/gateway/50_xml_server.xml)

NOTE

Because Q2 is running at another session, and monitoring the `deploy` directory for new service configurations, it is very important that the move of the XML file to deploy directory is atomic, so when it shows up there, it's a valid XML document.

If you open an editor, or use `curl` or `wget` to fetch the file from the jPOS server, it can happen that Q2 gets to see the file while it's being populated causing a failure. You'd see that failure in the log. If you download it to a `tmp` directory and then *move* it (using the `mv` command or `RENAME` if you're using a DOS) you should not have any problem.

Once `50_xml_server.xml` is deployed, the log will show something like this:

```
<log realm="xml-server-8000.server" at="2017-05-06T22:34:38.988" lifespan="61ms">
  <iso-server>
    listening on port 8000                                                    ①
  </iso-server>
</log>
```

① The server is listening to port 8000

TIP

If some other process is listening to port 8000, you'd see a self explanatory error like this:

```
<iso-server>
  <exception name="Address already in use (Bind failed)">
  ...
  ...
</iso-server>
```

Just change the port number in `50_xml_server.xml`.

You can use `netstat -an | grep LISTEN` to check that the server is properly listening.

While the system is running, open another terminal and call `telnet localhost 8000` (or `nc localhost 8000` if you have `netcat` installed):

On your terminal you'll see something like this:

```
$ telnet localhost 8000
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'. ①
```

① Use `Ctrl-]` to get back to telnet's command prompt (`telnet>`) where you can type `close` to close your session.

Then copy and paste the following XML-formatted 0800 message:

```
<isomsg>
 <field id="0" value="0800" />
 <field id="11" value="000001" />
 <field id="41" value="00000001" />
 <field id="70" value="301" />
</isomsg>
```

If you now switch to your Q2 session, you'll see the message in the log, something like this:

```
<log realm="xml-server-8000.server.session/127.0.0.1:54834" at="2017-05-
06T22:37:52.145">
  <session-start/>
</log>
<log realm="xml-server-8000/127.0.0.1:54837" at="2017-05-06T22:38:28.980"
lifespan="3240ms">
  <receive>
    <isomsg direction="incoming">
      <!-- org.jpos.iso.packager.XMLPackager -->
      <field id="0" value="0800"/>
      <field id="11" value="000001"/>
      <field id="41" value="00000001"/>
      <field id="70" value="301"/>
    </isomsg>
  </receive>
</log>
```

So far so good, the server has received the message and it has queued it into a Space queue called `TXNMGR`.

There's no TransactionManager running yet, so nothing happens and you don't get a response.

Configure a destination channel

Now add `10_channel_jpos.xml` to the deploy directory with the following content:

```
<channel-adaptor name='jpos-channel' class="org.jpos.q2.iso.ChannelAdaptor"
logger="Q2">
    <channel class="org.jpos.iso.channel.XMLChannel"
          packager="org.jpos.iso.packager.XMLPackager">
        <property name="host" value="isobridge.jpos.org" />                    ①
        <property name="port" value="9000" />                                  ②
    </channel>
    <in>jpos-send</in>                                                         ③
    <out>jpos-receive</out>                                                    ④
    <reconnect-delay>10000</reconnect-delay>
</channel-adaptor>
```

① jPOS auto-responder host

② jPOS auto-responder port

③ arbitrary name for incoming queue

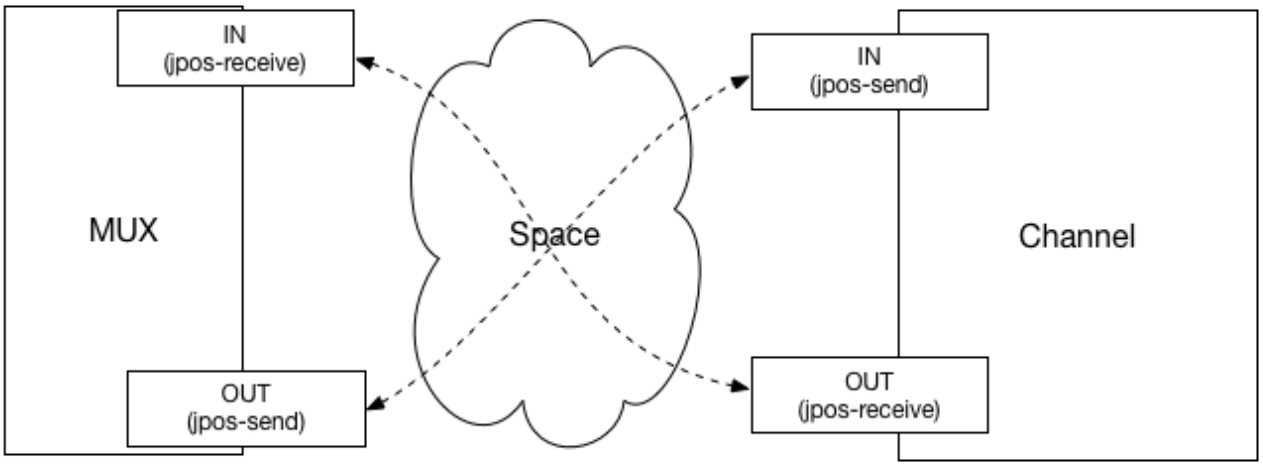
④ arbitrary name for outgoing queue

NOTE	We connect here to jPOS' auto-responder, which is usually up and listening on port 9000. You could use a local server, for example based on jPOS-EE's server simulator.

You can download the service configuration here: http://jpos.org/downloads/tutorials/gateway/10_channel_jpos.xml

TIP	In order to verify your connection to the jPOS auto-responder, you can use `netstat -an | grep 9000` and make sure it shows as `ESTABLISHED`. As an alternative, you can use to the jPOS console at http://jpos.org/jpos (username `admin`, password `test`) and navigate to `System→Log`.

Configure the multiplexer



In this gateway example, we have a persistent connection to the remote host but we can accept multiple connections to the incoming server. The server can process multiple transactions simultaneously and the responses to those transactions may arrive at different order (i.e. local cards may get a faster reply than an international card where the acquirer needs to

The multiplexer matches the responses to their original requests and allows us to know to which client socket (abstracted away as an `ISOSource`) we needs to provide a response. Please see the jPOS programmer's guide for details.

The MUX and Channels `<in>` and `<out>` queues are seen from the component's perspective. Think about connecting a a VCR to a TV, the VCR's `out` connection goes to the TV's `in` one, and vice-versa.

Create a file called `20_mux_jpos.xml` with the following content: (or download it from http://jpos.org/downloads/tutorials/gateway/20_mux_jpos.xml)

```
<mux class="org.jpos.q2.iso.QMUX" logger="Q2" name="jPOS-AUTORESPONDER">
    <in>jpos-receive</in>                                                    ①
    <out>jpos-send</out>                                                     ②
    <ready>jpos-channel.ready</ready>                                        ③
</mux>
```

① MUX's `in` is Channel's `out`

② And vice-versa

③ When a channel is connected to the remote host, it places an entry in the space using its service name (in this case `jpos-channel`) with the arbitrary suffix `.ready`.

Configure the TransactionManager

At this point we have a server listening to port 8000, a multiplexer and a channel connected to the remote jPOS auto-responder host.

We need to add the missing piece, the TransactionManager, to connect the dots between the incoming server and the outgoing channel.

This small transaction manager configuration doesn't add too much business logic, but this is the place where most jPOS applications implement the business requirements. A typical transaction manager would validate and sanitize an incoming message, perform dozens of validations (cards, terminals, acquirers, pin, CVV, velocity checks, balances) just by adding more participants to the XML configuration.

Create a file `30_txnmgr.xml` with the following content and place it in the `deploy` directory: (or download it from http://jpos.org/downloads/tutorials/gateway/30_txnmgr.xml)

```
<txnmgr class="org.jpos.transaction.TransactionManager" logger="Q2">
  <property name="queue" value="TXNMGR"/>                              ①
  <property name="sessions" value="2"/>
  <property name="max-sessions" value="128"/>
  <property name="debug" value="true"/>

  <participant class="org.jpos.transaction.participant.QueryHost"/>    ②
  <participant class="org.jpos.transaction.participant.SendResponse"/> ③
</txnmgr>
```

① We use the arbitrary queue name `TXNMGR` used in the server's ISORequestListener

② Query host participant takes care of querying the remote jPOS auto-responder host using the MUX

③ SendResponse will send us back a response == Testing the setup

Now use `telnet` one more time to connect to `localhost` on port `8000` and fire a 0800 message using copy & paste:

```
<isomsg>
 <field id="0" value="0800" />
 <field id="11" value="000001" />
 <field id="41" value="00000001" />
 <field id="70" value="301" />
</isomsg>
```

If everything worked OK, you should see a 0810 response.

If you go your Q2 terminal, you'd see something like this in the log:

```
<log realm="org.jpos.transaction.TransactionManager"
```

```
      at="2017-05-07T21:19:10.762" lifespan="192ms">
  <commit>
    txnmgr-1:idle:1
    <context>
      TIMESTAMP: Sun May 07 21:19:10 ART 2017
      PROFILER:
       <profiler>
         end [192.6/192.6]
       </profiler>

      SOURCE: org.jpos.iso.channel.XMLChannel@c7daa33
      REQUEST:
       <isomsg>
         <!-- org.jpos.iso.packager.XMLPackager -->
         <field id="0" value="0800"/>
         <field id="11" value="000001"/>
         <field id="41" value="00000001"/>
         <field id="70" value="301"/>
       </isomsg>

      DESTINATION: jPOS-AUTORESPONDER
      RESULT:
       <result/>

      :paused_transaction:
       id: 1

      RESPONSE:
       <isomsg>
         <!-- org.jpos.iso.packager.XMLPackager -->
         <field id="0" value="0810"/>
         <field id="11" value="000001"/>
         <field id="37" value="34219"/>
         <field id="38" value="599928"/>
         <field id="39" value="00"/>
         <field id="41" value="00000001"/>
         <field id="70" value="301"/>
       </isomsg>

    </context>
           prepare: org.jpos.transaction.participant.QueryHost PREPARED PAUSE
READONLY NO_JOIN
           prepare: org.jpos.transaction.participant.SendResponse PREPARED READONLY
            commit: org.jpos.transaction.participant.SendResponse
     in-transit=0, head=2, tail=2, outstanding=0, active-sessions=2/128, tps=0,
peak=0, avg=0.00, elapsed=192ms
    <profiler>
      prepare: org.jpos.transaction.participant.QueryHost [0.4/0.4]
      resume [181.5/181.9]
      prepare: org.jpos.transaction.participant.SendResponse [0.6/182.5]
       commit: org.jpos.transaction.participant.SendResponse [9.2/191.8]
```

```
      end [0.9/192.7]
    </profiler>
  </commit>
</log>
```

PART 2: Why it works

If we go back to our server configuration:

```
<server class="org.jpos.q2.iso.QServer" logger="Q2"
         name="xml-server-8000" realm="xml-server-8000">
    <attr name="port" type="java.lang.Integer">8000</attr>
    <channel class="org.jpos.iso.channel.XMLChannel"
             packager="org.jpos.iso.packager.XMLPackager"
             type="server"
             logger="Q2"
             realm="xml-server-8000">
        <property name="timeout" value="180000"/>
    </channel>
    <request-listener class="org.jpos.iso.IncomingListener" logger="Q2"    ①
                      realm="incoming-request-listener">
        <property name="queue"  value="TXNMGR" />                          ②
        <property name="ctx.DESTINATION"  value="jPOS-AUTORESPONDER" />    ③
    </request-listener>
</server>
```

① All incoming messages are handled by the IncomingListener ISORequestListener

② We tell IncomingListener to use `TXNMGR` as the forwarding queue name

③ We tell IncomingListener to place an arbitrary variable `DESTINATION` with a value `jPOS-AUTORESPONSE` in the context

You can see that we are using a standard request listener called IncomingListener, documented in the jPOS Programmer's Guide.

The `IncomingListener.process` method is quite simple:

```
public boolean process (ISOSource src, ISOMsg m) {
    final Context ctx  = new Context ();                                   ①
    ctx.put (timestamp, new Date());                                       ②
    ctx.getProfiler();                                                     ③
    ctx.put (source, src);                                                 ④
    ctx.put (request, m);                                                  ⑤
    if (additionalContextEntries != null) {
        additionalContextEntries.entrySet().forEach(                       ⑥
            e -> ctx.put(e.getKey(), e.getValue())
        );
    }
    sp.out(queue, ctx, timeout);                                           ⑦
    return true;
}
```

① We create a standard `org.jpos.transaction.Context` object.

② We place a timestamp `Date` in the Context, in case some TransactionParticipant cares to use it.

③ It creates a profiler object, in case it's needed.

④ We place a `source` variable with a reference to our `ISOSource` (this will be required by the `SendResponse` participant in order to send back a response.

⑤ We place the `request` object in the Context, so that the participants can use it (i.e. to query the remote host).

⑥ If the request listener's configuration object has entries starting with `"ctx."`, we place it in the context. This is how the `DESTINATION` variable is placed in the Context so that the `QueryHost` participant can get to know where to route the transaction.

⑦ We use the Space to queue the Context, in this case using the arbitrary `TXNMGR` queue.

The configuration of our IncomingListener is very short because we are taking advantage of several default variable names. If you take a look at the `setConfiguration` object you can easily get to know which are those defaults:

```
    public void setConfiguration (Configuration cfg)
        throws ConfigurationException
    {
        timeout  = cfg.getLong ("timeout", 15000L);                          ①
        sp = (Space<String,Context>) SpaceFactory.getSpace (cfg.get ("space")); ②
        queue = cfg.get ("queue", null);                                     ③
        if (queue == null)
            throw new ConfigurationException ("queue property not specified");
        source = cfg.get ("source", ContextConstants.SOURCE.toString());     ④
        request = cfg.get ("request", ContextConstants.REQUEST.toString());  ⑤
        timestamp = cfg.get ("timestamp", ContextConstants.TIMESTAMP.toString()); ⑥
        Map<String,String> m = new HashMap<>();
        cfg.keySet()
           .stream()
           .filter (s -> s.startsWith("ctx."))
           .forEach(s -> m.put(s.substring(4), cfg.get(s)));
        if (m.size() > 0)
            additionalContextEntries = m;
    }
```

① If no timeout is specified, we use a default of 15 seconds (15000 milliseconds).

② Unless a `space` configuration is present, we use the default Space (`tspace:default`).

③ `queue` has to be specified, otherwise we raise a `ConfigurationException`

④ `source` defaults to the literal `SOURCE` (this is why you see a `SOURCE` in the Debug output when you send a transaction).

⑤ `request` defaults to literal `REQUEST`

⑥ `timestamp` defaults to literal `TIMESTAMP`

NOTE	When the IncomingListener places an entry in the Space, the TransactionManager is supposed to pick it immediately. If for some reason it doesn't pick it, it is quite better to just let it expire than trying to handle it a later time, causing a snowball of reversals. This is why we have a 15 seconds default timeout. This timeout can be increased or even removed (by using a value of 0) if the TransactionManager is configured with an initial participant that actually checks the timestamp in the context and handles situations where the transaction is coming to it actually late, probably logging the transaction to an exception file and quickly getting ready to process the next transaction.

We use the Space to communicate between the ISORequestListener and the TransactionManager. We do this with the simple syntax `sp.out (queue, ctx);`.

This is a good time to review:

- The ISORequestListener API doc
- The Space chapter in the Programmer's Guide

`IncomingListener` pushes the Context to the Space under the `TXNMGR` queue and it gets picked up by the TransactionManager which has the following configuration:

```
<txnmgr class="org.jpos.transaction.TransactionManager" logger="Q2">
    <property name="queue" value="TXNMGR"/>                              ①
    <property name="sessions" value="2"/>                                ②
    <property name="max-sessions" value="128"/>                          ③
    <property name="debug" value="true"/>                                ④

    <participant class="org.jpos.transaction.participant.QueryHost"/>    ⑤
    <participant class="org.jpos.transaction.participant.SendResponse"/> ⑥
</txnmgr>
```

① The queue name, `TXNMGR`

② Default to two simultaneous sessions

③ Up to 128 sessions

④ This turns on the debug and trace, see this blog post

⑤ Our first participant, `QueryHost` will route the transaction to the `DESTINATION` remote host

⑥ Second participant, `SendResponse` will send back the response, if available, to the original `SOURCE`

Let's take look at the QueryHost code. We'll provide a simplified version here (mostly without error detection code).

```
public int prepare (long id, Serializable ser)  {
       Context ctx = (Context) ser;                                    ①
       ...
       ...
       String ds = ctx.getString(destination);                         ②
       ...
       String muxName = cfg.get ("mux." + ds , "mux." + ds);
       MUX mux =  (MUX) NameRegistrar.getIfExists (muxName);           ③
       ISOMsg m = (ISOMsg) ctx.get (requestName);
       ...
       if (isConnected(mux)) {
          mux.request(m, t, this, ctx);                                ④
          return PREPARED | READONLY | PAUSE | NO_JOIN;                ⑤
       } else {
           return result.fail(CMF.HOST_UNREACHABLE, ...)
       }
   }
```

① The TransactionParticipant interface does not mandate the use of a `Context` as the second parameter, it just has to be `Serializable`. We cast it to `Context`.

② We pick the `DESTINATION` placed by IncomingListener.

③ We locate a MUX with the destination's name.

④ We send the message through the MUX asynchronously.

⑤ We immediately return, without waiting for a response, that's why we use the `PAUSE` modifier.

When we return `PAUSE` the transaction manager suspends this transaction and it's ready to process next one. This is the reason why with a very low number of sessions you can process thousands of simultaneous transactions. If you look the code in `QueryHost`, it has a property called `continuations` that defaults to `true` but can be set to `false` to change its behaviour to synchronous mode. In that mode, we use the old `MUX.request(ISOMsg, timeout)` signature that blocks until a response arrives.

When using that mode, the transaction manager sessions have to be significantly raised in order to process a reasonable number of simultaneous transactions.

TIP It is trivial to add more intelligent routing to the system, you can add a participant before QueryHost so that it sets a proper `DESTINATION` in the context depending on the `REQUEST` message's BIN, amount, time of day, availability of remote connections, etc.

The final piece of the system is the SendResponse participant, it's code is very simple:

```
    private void sendResponse (long id, Context ctx) {
         ISOSource src = (ISOSource) ctx.get (source);
         ISOMsg resp = (ISOMsg) ctx.get (response);
         try {
             if (ctx.get (TX.toString()) != null) {
                 ctx.log("*** PANIC - TX not null - RESPONSE OMITTED ***");
             } else if (resp == null) {
                 ctx.log (response + " not present");
             } else if (src == null) {
                 ctx.log (source + " not present");
             } else if (!src.isConnected())
                 ctx.log (source + " is no longer connected");
             else
                 src.send (resp);
         } catch (Throwable t) {
             ctx.log(t);
         }
     }
```

Despite some validations and protections, it basically locates the original `ISOSource` placed by the `IncomingListener`, the `RESPONSE` placed by `QueryHost` participant, and calls `src.send(resp)` to send it back through the calling channel.

TIP

If you want to dig deeper into how this works, we suggest you carefully read the TransactionManager's documentation in the Programmer's Guide. You'll see that the `QueryHost` participant does it job at "prepare time" while the `SendResponse` does it at `commit/abort` time. Because the `SendResponse` participant may have to send a response even if an error occurs, it implements the `AbortParticipant` interface. This is something worth properly understanding.

Questions? e-mail support@jpos.org or join our Slack channel.