Loway
research

QueueMetrics
call center monitor

Accessing QueueMetrics through the XML-RPC interface

Version 1.7

January 14, 2009

Loway di Lorenzo Emilitri
Via Ligornetto 17a – 6854 San Pietro - Switzerland

Index

Document content

This document details how to access and use the XML-RPC access functionality in Loway QueueMetrics. This makes it possible for your programs to leverage the power of QueueMetrics by calling a very simple API, with bindings available in nearly every programming language.

Revision history

- Nov 13, 2006: First draft
- March 22, 2007: Added PHP example
- April 13, 2007: "raw" blocks available
- May 11, 2007: real-time blocks and auth server
- Jun 11, 2007: multi-stint calls
- Nov 11, 2007: enter-queue position and schedule adherence
- Nov 10, 2008: Added Outcomes data blocks and external XML-RPC Authentication description
- Jan 14, 2009: Added support for JavaScript

What is XML RPC?

Wikipedia defines XML-RPC as:

> *XML-RPC is a remote procedure call protocol which uses XML to encode its calls and HTTP as a transport mechanism. It is a very simple protocol, defining only a handful of data types and commands, and the entire description can be printed on two pages of paper. This is in stark contrast to most RPC systems, where the standards documents often run into the thousands of pages and require considerable software support in order to be used.*

This means that, whatever your programming language of choice, you can surely find an XML-RPC library for it; and once you have the library, connection to QueueMetrics with it is straightforward.

Which functions does QueueMetrics export as XML-RPC?

QueueMetrics exports the full results of an analysis in XML-RPC, so you can access whatever information you feel you may need. Information is divided into blocks, i.e. sets of data that correspond roughly to the tables QM uses for its own output.

This means that you can build software that sits on top of QueueMetrics and uses its result as a starting point for further elaboration, e.g.:

- Visualizing results with complex graphs currently not supported by QueueMetrics
- Computing period comparison analyses (one period versus another period)
- Accessing agent presence data for payroll computation

Of course there are a very big number of possible scenarios where you might want to use such information.

The XML-RPC interface is available in all version of QueueMetrics starting from version 1.3.1.

Example: accessing QueueMetrics from Python

In this example we'll see how easy it is to access QueueMetrics from a scripted language like Python. You can enter the following statements interactively using a Python IDE like IDLE, or make them a part of a larger program.

The following code connects to a the XML-RPC port of a QueueMetrics instance running at http://qmserver:8080/qm130 and asks for a couple of tables, namely the distribution of answered calls per day and the Disconnection causes.

```
> import xmlrpclib
> server_url = 'http://qmserver:8080/qm130/xmlrpc.do';
> server = xmlrpclib.Server(server_url);
> res = server.QM.stats( "queue-dps", "robot", "robot","", "",
"2005-10-10.10:23:12", "2007-10-10.10:23:10","", [
'KoDO.DiscCauses', 'CallDistrDO.AnsDistrPerDay' ] )

> res.keys()
['CallDistrDO.AnsDistrPerDay', 'result', 'KoDO.DiscCauses']

> res['result']
[['Status', 'OK'], ['Description', ''], ['Time elapsed (ms)',
3008], ['QM Version', '1.3.1']]

> res['result'][2][1]
3008
```

As you can see, it only takes four lines of Python code to connect to QueueMetrics and get all the results back!

Example: Accessing QueueMetrics from Java

This is an example functionally equivalent to the one above in Python, but it's written in Java using the Redstone XML-RPC client library.

```
import java.io.IOException;
import java.util.HashMap;
import java.net.URL;
import redstone.xmlrpc.XmlRpcClient;
import java.util.ArrayList;

public class xmlRpcTestClient {
    public void perform() {
        String stUrl = "http://server:8080/qm130/xmlrpc.do";
        System.setProperty(
                "org.xml.sax.driver",
                "org.apache.xerces.parsers.SAXParser"
        );

        try {
            XmlRpcClient client = new XmlRpcClient( stUrl, false );

            ArrayList arRes = new ArrayList();
            arRes.add( "OkDO.AgentsOnQueue" );
            arres.add( "KoDO.DiscCauses" );
            arRes.add( "KoDO.UnansByQueue" );
            arRes.add( "DetailsDO.CallsKO" );

            Object[] parms = { "queue-dps", "robot", "robot",
                               "", "", "2005-10-10.10:23:12",
                          "2007-10-10.10:23:10",
                               "", arRes };

            Object token = client.invoke( "QM.stats", parms );
            HashMap resp = (HashMap) token;
            System.out.println( "Resp: " + resp );
        } catch ( Exception e ) {
            e.printStackTrace();
        }
    }

    public static void main(String[] args) {
        xmlRpcTestClient c = new xmlRpcTestClient();
        try {
            c.perform();
        } catch (Exception e ) {
            e.printStackTrace();
        }
    }
}
```

We'll have to explicitly set which XML parser to use; for the rest, the code looks very much alike the Python one.

Example: Accessing QueueMetrics from PHP

This example is based on PHP's XML_RPC class, that is a part of PEAR and so should be preinstalled with any modern PHP installation.

```
<h1>A QueueMetrics XML-RPC client in PHP</h1>
<?
require_once 'XML/RPC.php';

$qm_server = "10.10.3.5";  // the QueueMetrics server address
$qm_port   = "8080";       // the port QueueMetrics is running on
$qm_webapp = "queuemetrics-1.3.3";  // the webapp name for
QueueMetrics


// set which response blocks we are looking for
$req_blocks = new XML_RPC_Value(array(
                new XML_RPC_Value("DetailsDO.CallsOK"),
                new XML_RPC_Value("DetailsDO.CallsKO")
           ), "array");

// general invocation parameters - see the documentation
$params = array(
new XML_RPC_Value("queue-dps"),
           new XML_RPC_Value("robot"),
           new XML_RPC_Value("robot"),
           new XML_RPC_Value(""),
           new XML_RPC_Value(""),
           new XML_RPC_Value("2007-01-01.10:23:12"),
           new XML_RPC_Value("2007-10-10.10:23:10"),
           new XML_RPC_Value(""),
           $req_blocks
        );

$msg = new XML_RPC_Message('QM.stats', $params);
$cli = new XML_RPC_Client("/$qm_webapp/xmlrpc.do", $qm_server,
$qm_port);
//$cli->setDebug(1);
$resp = $cli->send($msg);
if (!$resp) {
    echo 'Communication error: ' . $cli->errstr;
    exit;
}
if ($resp->faultCode()) {
    echo 'Fault Code: ' . $resp->faultCode() . "\n";
    echo 'Fault Reason: ' . $resp->faultString() . "\n";
} else {
    $val = $resp->value();
    $blocks = XML_RPC_decode($val);

    // now we print out the details....
    printBlock( "result", $blocks );
    printBlock( "DetailsDO.CallsOK", $blocks );
    printBlock( "DetailsDO.CallsKO", $blocks );
}

// output a response block as HTML
function printBlock( $blockname, $blocks ) {
     echo "<h2>Response block: $blockname </h2>";
     echo "<table border=1>";
     $block = $blocks[$blockname];
     for ( $r = 0; $r < sizeof( $block ); $r++ ) {
           echo "<tr>";
           for ( $c = 0; $c < sizeof( $block[$r] ); $c++ ) {
                 echo( "<td>" . $block[$r][$c] . "</td>" );
           }
           echo "</tr>\n";
     }
     echo "</table>";
}
?>
```

As you can see, it is very similar to the other programming languages, and reading into the results is simply a matter of selecting the correct block and then accessing the data cell by row and column. The added complexity is due to the explicit error condition check and result printout.

Example: Accessing QueueMetrics from JavaScript

In order to access the XML-RPC interface from JavaScript, you need to use an adaptor library. We have been able to use successfully a library called *JSON-XML-RPC* that can be found at http://code.google.com/p/json-xml-rpc/ in a file caller *rpc.js*

Generally speaking, a JavaScript client can make requests only against the same server that is serving the HTML page, therefore you need to install it on the same server as QueueMetrics, creating a separate webapp.

```
<html>
<head>
<title>javascript_client.html</title>
<script src="rpc.js"></script>
</head>
<body>
<h1>QueueMetrics JavaScript XML-RPC example </h1>
<script>
var server = "/DAILY/xmlrpc.do";

function run() {
  try {
    var service = new rpc.ServiceProxy( server, {
           asynchronous:false,
           protocol: "XML-RPC",
           methods: ["QM.stats", "QM.realtime", "QM.auth"]
    } );
    res = service.QM.stats( "q1", "robot", "robot","", "",
            "2005-10-10.10:23:12", "2009-10-10.10:23:10","",
            [ "KoDO.DiscCauses", "CallDistrDO.AnsDistrPerDay" ]
    );
    document.getElementById("RESULT").innerHTML = plotBlocks(res);
  } catch(e) {
    alert("Error raised: " + e);
  }
}


function plotBlocks( hmBlocks ) {
  s = "";
  for (var i in hmBlocks) {
    s += "<h2>Block: " + i + "</h2>";
    s += plotBlock( hmBlocks[i] );
  }
  return s;
}
```

```
function plotBlock( arBlock ) {
  s = "";
  for ( r =0; r < arBlock.length; r++ ) {
    s += "<tr>";
    for ( c = 0; c < arBlock[r].length; c++ ) {
      s += "<td>" + arBlock[r][c] + "</td>";
    }
    s += "</tr>";
  }
  return "<table border=1>"  + s + "</table>";
}
</script>

<input type="button" value="Click me!" onclick="run();" >
<div id="RESULT"></div>

</body>
</html>
```

As you can see, the code is actually very similar to the Python one. The only important difference here is that the names of the methods have to be explicitly declared.

Understanding call parameters

There are three methods exported by the XML-RPC interface, and they are used for three different reasons:

- **QM.stats**: get the main historical stats
- **QM.realtime**: get the real time stats (available since QM 1.3.5)
- **QM.auth**: use QueueMetrics as an auth server for third party software (available since QM 1.3.5)

The method QM.stats

This is the main methods exported, its XML-RPC name being *QM.stats*. It takes a number of arguments, all of which must be supplied and all of which must have the correct data type. They are:

1. *(String)* Queue name or names: the set of queues that must be include in the analysis. They must be separated by a "|" symbol if more than one queue is passed. The queue name is the internal Asterisk queue name.
2. *(String)* Access username: this one must be the user name of an active user holding the key ROBOT.

3. *(String)* Access password: this one must be the clear-text password of the given user name
4. *(String)* Logfile – always leave blank.
5. *(String)* Period – always leave blank.
6. *(String)* Start of period. This must be written in exactly the format *yyyy-mm-dd.hh.mm.ss* (do not forget the dot between the date and the hour).
7. *(String)* End of period. Same format as start of period
8. *(String)* Agent filter – an agent's name, like "Agent/101" that must be the filter for all the relevant activity
9. *(List)* A list of the required analyses to be returned to the client. Each analysis name must be supplied as a *String*.

This call will start up a session in QueueMetrics, check if the user exists and has the privilege to run the report, run the analysis, prepare the required results and return them. At the end of the call, the QueueMetrics session is destroyed so no data is kept for further elaboration.

This means that it's usually the most efficient thing to do to request all needed response information and once, but it's wise to limit yourself to the minimum data set you will actually need, as each block takes up CPU and memory space to be marshaled between the native Java format, the intermediate XML format and the resulting client format.

It is also advisable to run large data set analysis at night time or when nobody is accessing the system, as they may take quite a lot of RAM and CPU, and this may slow down QueueMetrics for the other live users.

The method QM.realtime

This method is very similar to *QM.stats* but it is used to retrieve the real time stats. It must be called with the following parameters:

1. *(String)* Queue name or names: the set of queues that must be include in the analysis. They must be separated by a "|" symbol if more than one queue is passed. The queue name is the internal Asterisk queue name.
2. *(String)* Access username: this one must be the user name of an active user holding the key ROBOT.
3. *(String)* Access password: this one must be the clear-text password of the given user name
4. *(String)* Logfile – always leave blank.
5. *(String)* Agent filter – an agent's name, like "Agent/101" that must be the filter for all the relevant activity

6. *(List)* A list of the required analyses to be returned to the client. Each analysis name must be supplied as a *String*.

The same suggestions that are given for *QM.stats* apply.

The method QM.auth

This method is used to authenticate a username / password couple against the QueueMetrics server. This can be used by third-party software that does not want to keep its own separate user database but wants to use QueueMetrics' instead.

1. *(String)* User name
2. *(String)* Password

There is only one response block returned, named "auth", where the caller will retrieve all user data, including the live key set for that user.

Understanding results

The result of the analyses is made up of a Struct, i.e. an associative array similar to Java's HashMap or Perl's Hash, that maps block names to blocks.

Each block is a rectangular data container made up of an Array of Arrays of Strings. The outer Array contains the rows while the inner ones will hold column values.

In the response, you get all the requested blocks as of parameter #9 of the call, plus a block named *response*. The block *response* will contain:

- The status of the call: this should always be OK if no error was encountered. If this is not OK, the other blocks may be missing or be meaningless.
- The version of QueueMetrics running in the server
- The time elapsed to run the report.

All other blocks will follow the convention that the inner Array #0 (the first row) will contain table headers, while actual data will be present from row 1 onwards. The table headers are the same you can see in the on-screen representation.

The following Python code will run a simple dump of all the results in a readable form:

```
for t in res.keys():
     print "===== Block " + t + ": ========"
     for r in range( len(res[t]) ):
          for c in range( len( res[t][r] )):
               print res[t][r][c] ,
               print "\t" ,
          print
```

The output will look like the following one:

```
===== Block result: ========
Status                 OK
Description
Time elapsed (ms)      3008
QM Version             1.3.1

===== Block KoDO.DiscCauses: ========
Cause                  N. Calls   &nbsp;     ...
Caller abandon         46         83.6%
Agent dump             1          1.8%
Timeout (voicemail)    8          14.5%
```

As you can see, the block *KoDO.DiscCauses* has one full line of header on its first row.

Using an external auth source for QueueMetrics

It is possible for QueueMetrics to accept an external authentication source. By using a simple XML-RPC interface, it is be possible to write easily third party scripts that are able to authenticate against a chosen authentication system (e.g. Kerberos, LDAP, a Microsoft domain server, etc).

Log on procedure

When an user logs in to QueueMetrics, QM checks to see if there is an XML-RPC auth source defined[1]. QueueMetrics will execute a query to that third-party server passing along the user-id and the password given for authentication, plus a given service-id that will be defined in the configuration file.

The server will respond passing along a fixed set of data:

- An auth response (see below)
- A set of login information for that user (e.g. real name, email…)
- The current class and user keys for that user

If the class passed along is empty, only the user keys will be computed. If the class is not empty, it will be searched by name on the QueueMetrics storage and it will be loaded. Trying to load a non-existent class will result in auth failure.

The auth server may return four different responses:

- Access allowed
- Access allowed with supplied user data
- Access is forbidden
- Access is fully delegated to QM

[1] If no external auth source is defined, the procedure will be the "old", pre-XML-RPC one. This is checked in the *default.authRpcServerUrl* system property.

The following table explains the relationship between those states:

	Auth OK	**Auth KO**
User data from RPC	AUTHORITATIVE	FORBIDDEN
User data from QM	SUCCESSFUL	DELEGATED

If auth was SUCCESSFUL, the local QM database is checked for that user name. If such a user is present, the user details, class and key information are loaded from QM. If such a user is not present, the details are taken from the ones supplied via XML-RPC.

If auth was AUTHORITATIVE, the details are taken from the ones supplied via XML-RPC. Then they are copied to the local user database (with a random password) so that though the user cannot login manually, it is possible to decode the user name for all logged operations (e.g. Updating a queue). If a user with the same name is present, its credentials are forcibly updated with the authoritative credentials.

If auth was FORBIDDEN, no other check is done and the user is rejected access.

If auth was DELEGATED, the standard QM logon procedure is done.

The whole procedure is wholly transparent to the user, that is they do not need to know which is the authority that grants or denies access.

XML-RPC call parameters

Method: QMAuth.auth

Input:

Pos	**Name**	**Type**	**Comment**
0	System ID	S	As set in QM; if not set: blank
1	Username	S	
2	Password	S	

Output:

Pos	Name	Type	Comment
0	Status code	S	One of: A (authoritative), S (success), F (forbidden), D (delegate). If other, behaves as D.
1	Real name	S	
2	Email	S	
3	Class name	S	Must match an existing class
4	User keys	S	

The following values are implied in QM:

- Enabled = yes
- Masterkey = no

The actual user data is only read by QM in case "A"; otherwise is ignored whatever is passed.

Tip: As a reference implementation, see the server that ships with QueueMetrics in the *mysql-utils/xml-rpc/xmlrpc_auth_server.php* file. It also contains an example of querying a LDAP server in PHP.

Configuration properties

A set of two new configuration properties is added in QM:

Property	Description
default.authRpcServerUrl	If set, XML-RPC auth is used. Points to the URL of the auth server.
default.authSystemId	The system-ID for this QM license. Can be currently any user-chosen name for the system.

Appendix I: Response block names

Each XML-RPC call will have its own admissible set of block names that can be asked for.

Response blocks come in families, also known as Data Objects. Each response block name is made up of the Data Object name dot the method name, as in *KoDO.DiscCauses.*

Remember that response block names are case sensitive.

Appendix I.1: Response block names for QM.stats

DO / Method	Description
AgentsDO	*Agent information*
AgentAvail	AGENT_AVAILABILITY
SessionPauseDur	SESSION_PAUSE_DUR
AnsCallsQueues	ANS_CALLS_FOR_QUEUES
AnsCallsSG	ANS_CALLS_FOR_SG
AnsCallsLocation	ANS_CALLS_BY_LOCATION
ReportAgents	REPORT_AGENTS
AreaAnDO	*Area analysis*
CallsOK	CALLS_OK
CallsKO	CALLS_KO
DistrDO	*ACD call distribution*
AcdByTerminals	ACD_BY_TERMINALS
AcdByQueue	ACD_BY_QUEUE

ReportAcd	REPORT_ACD
CallDistrDO	*Daily call distribution*
AnsDistrPerDay	ANS_DISTR_PER_DAY
AnsWaitPerDay	ANS_WAIT_PER_DAY
UnansWaitPerDay	UNANS_WAIT_PER_DAY
AnsDistrPerHr	ANS_DISTR_PER_HR
AnsWaitPerHr	ANS_WAIT_PER_HR
UnansWaitPerHr	UNANS_WAIT_PER_HR
AnsDistrPerDOW	ANS_DISTR_PER_DOW
AnsWaitPerDOW	ANS_WAIT_PER_DOW
UnansWaitPerDOW	UNANS_WAIT_PER_DOW
QPosPerDay	Queue position and length data *(since 1.4.3)*
QPosPerHr	Queue position and length data *(since 1.4.3)*
QPosPerDOW	Queue position and length data *(since 1.4.3)*
SalesPerDay	Sales data *(since 1.4.0)*
SalesPerHr	Sales data *(since 1.4.0)*
SalesPerDOW	Sales data *(since 1.4.0)*
StaffPerDay	Staff data *(since 1.4.3)*
StaffPerHr	Staff data *(since 1.4.3)*
StaffPerDOW	Staff data *(since 1.4.3)*
OkDO	*Taken calls*
AgentsOnQueue	AGENTS_ON_QUEUE

ServiceLevelAgreement	SERVICE_LEVEL_AGREEMENT
DisconnectionCauses	DISCONNECTION_CAUSES
Transfers	TRANSFERS
AnsweredcallsByQueue	ANSWERED_CALLS_BY_QUEUE
AnsweredcallsByDirection	ANSWERED_CALLS_BY_DIRECT
RiassAllCalls	RIASSUNTO_ALL_CALLS
RiassFullyWithin	RIASSUNTO_FULLY_WITHIN
StintsOk	Taken calls by number of stints
QPosOk	Queue position at enter for taken calls (since 1.4.3)
KoDO	*Lost calls*
DiscCauses	DISCONNECTION_CAUSES
UnansByQueue	UNANS_BY_QUEUE
UnansByLen	UNANS_BY_LENGTH
InclusiveSLA	INCLUSIVE_SLA
ReportKoAll	REPORT_KO_ALL
ReportKoFully	REPORT_KO_FULLY
StintsKo	Lost calls by number of stints
StintsOkKo	All calls by number of stints
QPosKo	Queue position at enter for lost calls *(since 1.4.3)*
QPosOkKo	Queue position at enter for all calls *(since 1.4.3)*
OutcomesDO	*Call outcomes*
GeneralRep	The top report

CallResByOutcome	Call results by outcomes
ActivBillable	Billable activities
ActivNotBillable	Non-billable activities
AgentReportDetailed	Detailed agent report
DetailsDO	*Full call details*
CallsOK	CALLS_OK
CallsKO	CALLS_KO
AgentSessions	AGENT_SESSIONS
CallsOkRaw *(since 1.3.4)*	Raw detail of all call information for taken calls
CallsKoRaw *(since 1.3.4)*	Raw detail of all call information for lost calls
AgentSessionsRaw *(since 1.3.4)*	Raw detail of all agent session information

Appendix I.2: Response block names for QM.realtime

DO / Method	Description
RealtimeDO	*Real time information*
RTRiassunto	AGENT_AVAILABILITY
RTCallsBeingProc	SESSION_PAUSE_DUR
RTAgentsLoggedIn	ANS_CALLS_FOR_QUEUES

Appendix I.3: Response block names for QM.auth

The *QM.auth* call will return one single block named “auth”.

This block contains the following information:

- *UserName*: the login name
- *Status*: OK if authentication was passed or ERR if it was not passed
- *FullName*: The user’s full name
- *Email*: The user’s email address
- *Class*: The name of the class the user belongs to
- *Keys*: The active key set of the user, that is, all keys given to the class plus or minus the keys that have been granted or revoked to this specifical user
- *Masterkey*: If set to 1, this user has a Masterkey, that is this user will pass each key ckeck
- *NLogons*: The number of logons the user has made. Each successful *QM.auth* call counts as a logon.

Appendix II: A short list of XML-RPC libraries

To access QueueMetrics, you only need a library with Client capabilities. Server capabilities are not needed. The following list is by no means exhaustive of all possible implementations available:

Language	**Libraries**
Perl	The BlackPerl library is available at http://www.blackperl.com/RPC::XML/
Python	The *xmlrpclib* ships with any modern version of the language
JavaScript	The JSON-XML-RPC library can be found at: http://code.google.com/p/json-xml-rpc/ Also, the Jsolait library offers an XML-RPC module: http://jsolait.net/
Java	There are a lot of implementations available for Java, we recommend Redstone's LGPL library - http://xmlrpc.sourceforge.net/
C / C++	See http://xmlrpc-c.sourceforge.net/
C# / .Net	A connector is available at http://www.xml-rpc.net/
VisualBasic	A COM component that will work on most languages on the Windows platform: http://sourceforge.net/projects/comxmlrpc
PHP	The package XML_RPC is a part of the standard Pear library.