vms Software

LibRabbitMQ V2.6-0 for VSI OpenVMS Alpha and Integrity

August 2019

Introduction

Thank you for your interest in LibRabbitMQ for VSI OpenVMS. LibRabbitMQ provides an API that can be used by OpenVMS-based software applications to exchange data with the RabbitMQ message broker (http://www.rabbitmq.com) via the Advanced Message Queuing Protocol (AMQP). LibRabbitMQ for VSI OpenVMS is based on the Open Source rabbitmq-c API (https://github.com/alanxz/rabbitmq-c) and can be used with most 3GL programming languages available for OpenVMS, including C/C++, FORTRAN, COBOL, BASIC, and Pascal.

Acknowledgements

VMS Software Inc. would like to acknowledge the support and assistance of the RabbitMQ community and their ongoing efforts with regard to developing and supporting the rabbitmq-c Open Source software package.

Requirements

The kit you are receiving has been compiled, built, and tested using the operating system and compiler versions listed below (or comparable). While it is highly likely that you will have no problems installing and using the kit on systems running higher versions of the products listed here, we cannot say for sure that you will be so lucky if your system is running older versions.

- OpenVMS 8.4-1H1 or higher
- VSI TCP/IP, HPE TCP/IP Services for OpenVMS, or MultiNet TCP/IP stack for network communication
- OpenSSL 1.1.1 (statically linked into the supplied LibRabbitMQ shareable image)

  Note that if you wish to statically link application code requiring with the supplied object libraries and require SSL/TLS support, it will be necessary to link with a comparable OpenSSL distribution.

In addition to the software requirements and options listed above, you will require language compilers for the programming languages that you intend to use.

It is assumed that the reader has a good knowledge of OpenVMS and software development in the OpenVMS environment.

vms Software

Recommended reading

It is recommended that application developers read some of the tutorials and other excellent documentation available on the RabbitMQ web site (http://www.rabbitmq.com). Developers should also be sure to and examine end experiment with the samples programs provided with LibRabbitMQ for VSI OpenVMS. You might also wish to join the RabbitMQ user mailing list (see https://groups.google.com/forum/?hl=en#!forum/rabbitmq-users), which provides access to the core RabbitMQ development team, and a thriving, knowledgeable, and helpful community of other RabbitMQ users.

Installing the kit

The kit is provided as an OpenVMS PCSI kit (`VSI-I64VMS-LIBRABBITMQ-V0206-0-1.PCSI` for I64 or `VSI-AXPVMS-LIBRABBITMQ-V0206-0-1.PCSI` for Alpha) that can be installed by a suitably privileged user using the following command:

```
$ PRODUCT INSTALL LIBRABBITMQ
```

The installation will then proceed as follows (output may differ slightly from that shown depending on platform and other factors):

```
Performing product kit validation of signed kits ...

The following product has been selected:
    VSI AXPVMS LIBRABBITMQ V2.6-0          Layered Product

Do you want to continue? [YES]

Configuration phase starting ...

You will be asked to choose options, if any, for each selected
product and
For any products that may be installed to satisfy software dependency
requirements.

Configuring VSI AXPVMS LIBRABBITMQ V2.6-0

    VMS Software Inc.

* This product does not have any configuration options.

Execution phase starting ...

The following product will be installed to destination:
    VSI AXPVMS LIBRABBITMQ V2.6-0
DISK$FUNYET_SYS:[VMS$COMMON.]

Portion done: 0%...40%...50%...80%...90%

...100%IT (queue SYS$BATCH, entry 63) completed

The following product has been installed:
    VSI AXPVMS LIBRABBITMQ V2.6-0          Layered Product

VSI AXPVMS LIBRABBITMQ V2.6-0
```

```
Post-installation tasks are required.


To start LibRabbitMQ at system boot time, add the following lines
to SYS$MANAGER:SYSTARTUP_VMS.COM:

    $ file := SYS$STARTUP:LIBRABBITMQ$STARTUP.COM
    $ if f$search("''file'") .nes. "" then @'file'

To stop LibRabbitMQ at system shutdown, add the following lines
to SYS$MANAGER:SYSHUTDWN.COM:

    $ file := SYS$STARTUP:LIBRABBITMQ$SHUTDOWN.COM
    $ if f$search("''file'") .nes. "" then @'file'
```

Post-installation steps

After the installation has successfully completed, include the commands displayed at the end of the installation procedure into `SYSTARTUP_VMS.COM` to ensure that the logical names required in order for users to use the software are defined system-wide at start-up.

In addition to the system logical name `LIBRABBITMQ$ROOT` (which points to root directory of the LibRabbitMQ installation tree), the logical name `LIBRABBITMQ$SHR` is also defined. This logical name points to the shareable image `LIBPQ$ROOT:[LIB]LIBPQ$SHR.EXE`, which can be linked with application code. Alternatively, it is possible to statically link application code with the object libraries found in the `LIBRABBITMQ$ROOT:[LIB]` directory.

Privileges and quotas

Generally speaking there are no special quota or privilege requirements for applications developed using LibRabbitMQ, although a reasonably high `BYTLM` is recommended, particularly if applications will transfer large amounts of data. The following quotas should be more than adequate for most purposes:

```
Maxjobs:          0  Fillm:       256  Bytlm:       128000
Maxacctjobs:      0  Shrfillm:      0  Pbytlm:           0
Maxdetach:        0  BIOlm:       150  JTquota:       4096
Prclm:           50  DIOlm:       150  WSdef:         4096
Prio:             4  ASTlm:       300  WSquo:         8192
Queprio:          4  TQElm:       100  WSextent:     16384
CPU:         (none)  Enqlm:      4000  Pgflquo:     256000
```

Installing in an alternative location

By default the software will be installed in `SYS$SYSDEVICE:[VMS$COMMON]`. If you wish to install the software in an alternative location this can be achieved using the `/DESTINATION` qualifier with the `PRODUCT INSTALL` command to specify the desired location; however it is important to note that an additional manual step will then be required to complete the installation. Specifically, when an alternative destination is specified, the start-up and shutdown procedures (`LIBRABBITMQ$STARTUP.COM` and `LIBRABBITMQ$SHUTDOWN.COM`)

will be placed into a subdirectory `[.SYS$STARTUP]` residing under the specified destination directory. If you wish to run these files from your standard `SYS$STARTUP` directory they will need to be copied from the destination subdirectory into your systems `SYS$STARTUP` directory.

Sample applications

The directory `LIBRABBITMQ$ROOT:[EXAMPLES]` contains several simple example programs written in C, COBOL, FORTRAN, and BASIC that serve to illustrate the usage of the LibRabbitMQ API. There is a command procedure `EXAMPLES.COM` that may be used to build the examples. These examples are intended to provide an introduction to the API and to hopefully serve as a basis for the development of more sophisticated applications.

The following notes provide a brief overview of some of the example programs and how to run them. Note that some examples have the TCP/IP address and port number of the RabbitMQ broker and username/password details hard-wired and these values will need to be changed as appropriate to reflect your environment. After modifying the port number and TCP/IP address details to reflect your environment, execute the commend procedure `EXAMPLES.COM` to compile and link the example code.

Example	Relevant files	Notes
FORTRAN producer, COBOL consumer	• `FOR_PRODUCER.EXE` • `COB_CONSUMER.EXE`	A pair of programs that illustrate basic producer/consumer functionality. The producer publishes messages that are routed into "test queue" and the consumer consumes messages from this queue. The producer publishes 1000 messages of size 2^n where $1 \leq n \leq 15$ and displays the time taken to publish each batch of messages. Note that the COBOL consumer specifies a pre-fetch count using the `AMQP$BASIC_QOS()` function. You may wish to experiment with pre-fetch count and to see how changing this value impacts consumer performance and memory usage.
Request-response example	• `RR_CLIENT.EXE` • `UARS.EXE` • `UARS.COM`	The request/response example illustrates how the API can be used to implement pseudo-synchronous request-response processing using AMQP. This functionality is provided

		by the AMQP$CALL() utility function, which is called by the client to send the request buffer and receive the response. The server is implemented using the generic server, AMQP$SERVER.EXE (run from UARS.COM), which greatly simplifies the server-side development effort. The client issues 10000 calls and displays the transaction rate. Requests are published by RR_CLIENT.EXE to the amq.direct exchange with routing key "SVC4", and this key is mapped by AMQP$SERVER.EXE to the function MY_SVC4 in the shareable image UARS.EXE. The function AMQP$CALL() establishes a unique and exclusive reply queue to which responses are written.
C producer (enqueue example)	• ENQUEUE.EXE • ENQUEUE-PERSIST.EXE • BAS_DEMO.BAS • UARS.EXE • UARS.COM	This example illustrates the use of AMQP$SERVER.EXE as a generic consumer. Messages are published by ENQUEUE.EXE (or ENQUEUE-PERSIST.EXE) to the amq.direct exchange using the routing key "SVC1", and this key is mapped by AMQP$SERVER.EXE to the function MY_SVC1 in the shareable image UARS.EXE. Messages published by ENQUEUE-PERSIST.EXE are published with delivery-mode 2, which means that messages will be persisted to disk and will not be lost if the broker is restarted before the messages are consumed, assuming that the queue into which they are routed is also durable. Messages can also be published to "SVC1" using the BASIC example code, BAS_DEMO.BAS.

Multi-threaded consumer	• `THREADS.EXE`	A simple multi-threaded consumer that is implemented using a new experimental API. The new API is intended to simplify client development without imposing too many restrictions; this API will be evolved and documented in subsequent releases of LibRabbitMQ for VSI OpenVMS. The example consumer establishes two connections to the RabbitMQ broker and each connection consumes from a single queue (via a single channel). The function `RabbitMQ_serve_thread ()` is then used to start a separate consumer thread for each connection, and for each message received the specified callback function (`callback_1()` or `callback_2()`) will be invoked. As currently implemented, the threads will terminate only if an error is encountered; this behavior may be changed in future releases to facilitate clean shutdown of processes. Before running this example, you should ensure that queues named "`foo`" and "`baa`" are created and are bound to the `amq.direct` exchange (or another direct exchange) with binding keys of "`foo`" and "`baa`" respectively. To test the example, you may then publish messages to the `amq.direct` exchange with routing keys of "`foo`" and "`baa`" and observe that the messages are consumed by the different threads.

Assuming that you have modified the examples (including the file UARS.COM) to specify the appropriate location of the RabbitMQ broker, the examples described in the table above may be run as follows:

- **Running the FORTRAN producer/COBOL consumer example**

  1. In one OpenVMS session, start the consumer:

     ```
     $ run cob_consumer.exe
     ```

  2. Open a second OpenVMS session and run the producer:

     ```
     $ run for_producer.exe
     ```

     Once the consumer starts reading messages off the queue it will output a progress counter for every 1000 messages read. After publishing each set of 100000 messages to the queue, the producer will report the time taken to publish the set of messages and the number of messages published per second. The producer will terminate after publishing the 16 sets of messages; the consumer continues to listen for messages indefinitely and must be terminated by entering CNTRL-Y.

- **Running the request-response example with AMQP$SERVER.EXE**

  1. In one OpenVMS session execute the UARS.COM command procedure, specifying as parameters the TCP/IP address (or host name) and port number for the RabbitMQ broker (replace the TCP/IP address and port number specified here with values applicable to your environment):

     ```
     $ @uars.com 16.156.32.108 5672
     ```

     The UARS.COM command procedure runs the generic server AMQP$SERVER.EXE, which loads the shareable image UARS.EXE and maps routing keys to function names in the shareable image as specified via the -s command line option. Multiple mappings may be specified using the -s option as illustrated in UARS.COM.

  2. Open another OpenVMS session and run the request-response client:

     ```
     $ run rr_client.exe
     ```

     The client publishes messages using routing key "SVC4", which is mapped by AMQP$SERVER.EXE to the *user action routine* "my_svc4" in UARS.COB. The user action routine routes responses back to the client using a unique reply queue specific (and exclusive) to the client in question. Upon completion the client displays the number of roundtrip calls processed per second.

- **Running the C producer example**

  1. In one OpenVMS session execute the UARS.COM command procedure, specifying as parameters the TCP/IP address (or host name) and port number for the RabbitMQ broker:

     ```
     $ @uars.com 16.156.32.108 5672
     ```

2. Open another OpenVMS session and run the C producer (ENQUEUE.EXE or ENQUEUE-PERSIST.EXE):

   ```
   $ run enqueue.exe
   ```

   The producer publishes messages to the amq.direct exchange using the routing key "SVC1", which is mapped by AMQP$SERVER.EXE to the *user action routine* "my_svc1" in UARS.COB. The user action routine displays the text "Hello from SVC1" for each message that is successfully read from the queue by AMQP$SERVER.EXE and passed to the action routine. Unlike the request-response example above, the user action routine does not return a reply message[1].

- **Modifying the C producer example to publish to multiple consumers**

  The above examples all operate in a point-to-point fashion, using the direct exchange amq.direct to publish each message to an individual consumer instance. The previous example can be easily modified as described below to publish messages to multiple consumers by instead using the topic exchange amq.topic:

  1. Edit enqueue.c (or enqueue-persist.c) and change the name of the exchange from "amq.direct" to "amq.topic". Save your changes and rebuild enqueue.exe (or enqueue-persist.exe) by re-running examples.com.
  2. Edit UARS.COM and include the following option to instruct amqp$server.exe to bind to the "amq.topic" exchange:

     ```
     "-e" "amq.topic"
     ```

  3. If you now start two or more consumers by running multiple instances of UARS.COM, and then run enqueue.exe (or enqueue-persist.exe), you should see that all consumers receive a copy of each message.

Tcl scripting engine (BUGS.EXE)

This release of LibRabbitMQ for VSI OpenVMS includes a Tcl-based scripting utility named BUGS.EXE with language extensions RabbitMQ. This scripting tool can be useful for prototyping and testing.

At this time the language extensions are not documented (it is hoped that documentation will be available for inclusion in the next release); however several simple examples are included with this kit to illustrate the capabilities of this scripting facility. The examples are described below and the code for these examples may be found in the examples directory librabbitmq$root:[examples.tcl].

Script	Notes
consumer.tcl	A simple consumer script that declares an auto-delete (temporary) queue (with a randomly generated name) and binds it to the built-

[1] AMQP$SERVER.EXE currently acknowledges successfully consumed messages. An option to disable acknowledgements may be provided in future releases.

	in direct exchange "amq.direct" with binding key "tcl-test". Any messages published to the "amq.direct" exchange with a routing key of "tcl-test" will be routed into the queue and will be received and displayed by the consumer.
get.tcl	A trivial example script that declares an auto-delete queue named "get-test", publishes a message into this queue via the default exchange (""), and gets the message from the queue. The example is intended to illustrate the use of the AMQP "basic.get" method, which can be used to explicitly get the next message (if available) from the specified queue, as opposed to the AMQP "basic.consume" method, where the broker effectively pushes messages down to the client (consumer) as fast as it is permitted.
producer.tcl	This example publishes a large number of messages to the "amq.direct" exchange with a routing key of "tcl-test" and may be used in conjunction with either the comsume.tcl example or the service.tcl example.
props.tcl	A trivial example that illustrates how to allocate, populate, and delete properties structures that can be used to specify message properties when publishing messages. The code fragment does not perform any AMQP operations. Note that only a subset of the message properties defined by the AMQP standard are currently supported by the Tcl interface. The supported properties are delivery mode (-delivery-mode), content type (-content-type), content encoding (-content-encoding) and the specification of one or more arbitrary headers (-headers).
rpc.tcl	An RPC client script that can be used in place of rr_client.exe (refer to the request-response example above in the section *Sample LibRabbitMQ AMQP applications*) to demonstrate RPC-style functionality with amqp$server.exe. Instructions on running this example are provided below.
serve.tcl	This example illustrates the specification and use of callback functions to receive and process messages based on binding keys. The script declares an auto-delete queue (with a RabbitMQ-generated name) and uses the RMQ::register command to bind the queue to the amq.direct exchange with a binding key of "tcl-test" and to associate the procedure TESTPROC with this binding. The procedure TESTPROC will then be called for any messages consumed that were published with a routing key of "tcl-test". Any number of callback functions can be registered, and the same callback function can be specified for different binding keys (however bindings must be unique). The command RMQ::serve listens for (consumes) messages and invokes the relevant callback function (if any) to process each message.
spy.tcl	This example illustrates the RMQ::spy command, which registers a consumer on the logging exchange amq.rabbitmq.log and

	associates with that consumer a procedure to process any consumed messages. RabbitMQ publishes its log file entries to this `amq.rabbitmq.log` topic exchange, using the severity level of the log messages as the routing key. By consuming from a queue (or queues) bound to this exchange with appropriate bindings it is therefore possible to monitor in real-time broker activity. The `RMQ::spy` command simplifies the implementation of such a monitoring facility into a single command. The option "`-all`" causes log messages of any severity to be consumed; specific severities can be specified using `-info`, `-warning`, or `-error` instead of `-all`. Note that it is possible to specify a callback procedure with the `RMQ::spy` command that will be called for each message consumed; if no callback procedure is specified, messages will simply be displayed to `SYS$ERROR`. Multiple `RMQ::spy` commands may be specified to consume and process log messages of different severity using different callback functions. Use of the `RMQ::spy` command is further described below.

Assuming that you have modified the Tcl examples `RPC.TCL` and `SPY.TCL` to correctly specify the location of your RabbitMQ broker and the broker is running, these examples may be run as follows:

- **Running the RPC.TCL example**

  1. In one OpenVMS session execute the `UARS.COM` command procedure (assuming that you have previously built this example as per the instructions in *Sample libRabbitMQ AMQP applications*), specifying as parameters the TCP/IP address (or host name) and port number for the RabbitMQ broker (replace the TCP/IP address and port number specified here with values applicable to your environment):

     ```
     $ @uars.com 16.156.32.82 5672
     ```

  2. Open another OpenVMS session and use the Tcl script interpreter to run the `RPC.TCL` client:

     ```
     $ bugs :== $librabbitmq$root:[bin]bugs.exe
     $ bugs rpc.tcl
     ```

     If all is well, the RPC client will display the text "`Hello there`", which is the response message from the RPC server (the procedure `MY_SVC4` in `UARS.COB`).

- **Running the SPY.TCL example**

  1. In one OpenVMS session, define a foreign command for the Tcl scripting engine and run the `SPY.TCL` script:

     ```
     $ bugs :== $librabbitmq$root:[bin]bugs.exe
     $ bugs spy.tcl
     ```

     Assuming that script successfully connects to the broker, it will now sit idle, waiting for log messages.

2. In another OpenVMS session, run `BUGS.EXE` and use the `RMQ::attach` command to connect to the RabbitMQ broker, replacing the address specified here with the correct address of your broker, and then use `CNTRL-Z` to exit the script interpreter:

```
$ run librabbitmq$root:[bin]bugs.exe
bugs> set ch [RMQ::attach "amqp://16.156.32.82:5672"]
bugs> ^Z
```

Upon establishing the connection, the `SPY.TCL` script will receive and output a message indicating that the broker has accepted a new connection, and upon exiting the script interpreter by entering `CNTRL-Z` the `SPY.TCL` script will display a message indicating that a connection to the broker is being closed. Specially, the output will be similar to the following (with different address details):

```
accepting AMQP connection <0.6298.5> (16.156.32.108:64713 ->
16.156.32.82:5672)
closing AMQP connection <0.6298.5> (16.156.32.108:64713 ->
16.156.32.82:5672):
connection_closed_abruptly
```

Note that the "`connection_closed_abruptly`" message is displayed because the interpreter session was terminated without explicitly closing the AMQP connection; this message is innocuous.

What's missing?

The supplied kit for OpenVMS includes all functionality supported by the Open Source rabbitmq-c client API. In addition, the port includes a language-agnostic API that makes it straightforward to write RabbitMQ applications using 3GL languages such as COBOL and FORTRAN. However, it should be noted that the language-agnostic API currently does not support the specification of all message properties. It is anticipated that this limitation will be addressed in future releases of the software.

Known problems and limitations

- The language-agnostic OpenVMS API implementation does not support the specification of all message properties. It is anticipated that this limitation will be addressed in future releases of the API.
- To ensure correct parsing of command line arguments when using `AMQP$SERVER.EXE` it is recommended that users set the process parse style to "extended" ("`set process/parse_style=extended`") or enclose command line arguments and options in double quotes.