ZeroMQInterface

ZeroMQ bindings for GAP

0.15

1 July 2024

Markus Pfeiffer

Reimer Behrends

Markus Pfeiffer
Email: markus.pfeiffer@st-andrews.ac.uk
Homepage: http://www.morphism.de/~markusp/
Address: School of Computer Science
University of St Andrews
Jack Cole Building, North Haugh
St Andrews, Fife, KY16 9SX
United Kingdom

Reimer Behrends
Email: behrends@gmail.com
Homepage: http://www.mathematik.uni-kl.de/agag/mitglieder/wissenschaftliche-mitarbeiter/dr-reimer-be
Address: Technische Universität Kaiserslautern
Fachbereich Mathematik
Postfach 3049
67653 Kaiserslautern
Deutschland

Copyright

Acknowledgements

We appreciate very much all past and future comments, suggestions and contributions to this package and its documentation provided by GAP users and developers.

Contents

1 Introduction **4**
- 1.1 Purpose and goals of this package . . . 4
- 1.2 Overview over this manual . . . 4
- 1.3 Installation . . . 4
- 1.4 Feedback . . . 4

2 ZeroMQ Bindings **5**
- 2.1 Addresses, transports, and URIs . . . 5
- 2.2 Creating and closing sockets . . . 6
- 2.3 Binding and connecting sockets to addresses . . . 8
- 2.4 Sending and receiving messages . . . 9
- 2.5 Setting and querying socket properties . . . 11

3 Using ZeroMQ with the zgap script **14**
- 3.1 Running zgap . . . 14
- 3.2 Zgap API . . . 15

Index **18**

Chapter 1

Introduction

1.1 Purpose and goals of this package

This package provides low-level bindings to the popular ZeroMQ distributed messaging framework for GAP and HPC-GAP as well as some higher level functions to ease the use of distributed messaging in GAP

1.2 Overview over this manual

Chapter 2 gives an overview of the available bindings, and examples of how to use them. Chapter 3 showcases one way of using ZeroMQInterface.

1.3 Installation

To use this package ZeroMQ needs to be installed on your system and the ZeroMQInterface package itself needs to be compiled. To install ZeroMQ please refer to its installation instructions. To compile this package, inside its root directory run:

Example

```
> ./configure
> make
```

Or

Example

```
> ./configure --with-zmq=path-to-your-zeromq
> make
```

to specify where ZeroMQ is located in you system.

1.4 Feedback

For bug reports, feature requests and suggestions, please use our issue tracker.

Chapter 2

ZeroMQ Bindings

There are experimental bindings to the ZeroMQ library available http://www.zeromq.net/. This section describes these bindings. Messages in ZeroMQ are sent between endpoints called *sockets*. Each socket can be *bound* to an address specified by a URI and other sockets can *connect* to the same address to exchange messages with that socket.

2.1 Addresses, transports, and URIs

Addresses are specified as URIs of one of four different types (TCP, IPC, in-process, PGM/EPGM), each for a different type of transport.

2.1.1 The TCP transport

TCP URIs map to POSIX TCP stream sockets. The URI is of the form `tcp://<address>:<port>` or `tcp://*:<port>`. Here, `address` is an internet address, either an IP address or a symbolic address (note that to resolve symbolic addresses, the library may have to consult DNS servers, which can take an indefinite amount of time or even fail). Port is a TCP port number. If a "*" is given instead of an address, this describes the so-called unspecified address; the URI can only be used for binding and will then accept incoming connections from all interfaces (as in binding to "0.0.0.0" in IPv4 or "::" in IPv6).

2.1.2 The IPC transport

The URI for IPC communication is of the form `ipc://<path>`, where `path` is an actual path on the file system. Binding to such a URI will create a file in that location.

Example

```
gap> socket := ZmqDealerSocket();;
gap> ZmqBind(socket, "ipc:///tmp/connector");
```

2.1.3 The in-process transport

The in-process transport is used to communicate between threads in order to avoid the overhead of operating system calls. Messages are simply being copied from one thread's memory to the other's. In-process URIs are of the form `inproc://<string>`, where `string` is an arbitrary string.

2.2 Creating and closing sockets

Sockets are generally being created via calls to `ZmqPushSocket` (2.2.1), etc. Each such call takes two optional arguments, a URI and an identity. If a URI is given, a call to `ZmqAttach` (2.3.3) will be performed immediately with the socket and URI. In particular, if the URI is prefixed with a "+" character, then the socket will connect to the address specified by the part after the "+" character; otherwise, it will be bound to the URI.

Example

```
gap> z := ZmqPushSocket("inproc://test");  # binds to inproc://test
gap> z := ZmqPushSocket("+inproc://test"); # connects to inproc://test
```

If an identity is also provided, the library will call `ZmqSetIdentity` (2.5.1) to set the identity (name) for that socket. For a precise description of the behavior of each socket type, please consult the original ZeroMQ documentation for `zmq_socket()`.

2.2.1 ZmqPushSocket

▷ `ZmqPushSocket(`*`[uri[, identity]]`*`)` (function)

A push socket is one end of a unidirectional pipe. Programs can send messages to it, which will be delivered to a matched pull socket at the other end.

2.2.2 ZmqPullSocket

▷ `ZmqPullSocket(`*`[uri[, identity]]`*`)` (function)

A pull socket is the other end of a unidirectional pipe.

2.2.3 ZmqReplySocket

▷ `ZmqReplySocket(`*`[uri[, identity]]`*`)` (function)

A reply socket provides the server side of a remote-procedure call interaction. It alternates between receiving a message and sending a message to the socket from which the previous one originated. Deviating from that protocol (for example, by sending two messages in succession or receiving two without responding to the first) will result in an error.

2.2.4 ZmqRequestSocket

▷ `ZmqRequestSocket(`*`[uri[, identity]]`*`)` (function)

A request socket provides the client side of a remote-procedure call interaction. It will alternate between sending a message to a connected reply socket and receiving the response.

2.2.5 ZmqPublisherSocket

▷ `ZmqPublisherSocket(`*`[uri[, identity]]`*`)` (function)

A publisher socket is a unidirectional broadcast facility. It will send each outgoing message to all connected subscriber sockets.

2.2.6 ZmqSubscriberSocket

▷ `ZmqSubscriberSocket(`*`[uri[, identity]]`*`)` (function)

A subscriber socket receives messages from a publisher socket. It can subscribe to only a specific subseet of messages (see the `ZmqSubscribe` (2.5.11) function) or receive all of them.

2.2.7 ZmqDealerSocket

▷ `ZmqDealerSocket(`*`[uri[, identity]]`*`)` (function)

A dealer socket is a bidirectional socket. One or more peers can connect to it. Outgoing messages will be sent to those peers in a round-robin fashion (i.e., the first message goes to the first peer, the second to the second peer, and so forth until all peers have received a message and the process begins anew with the first peer). Incoming messages will be received from all peers and processed fairly (i.e., no message will be held indefinitely). Two dealer sockets can be used to create a bidirectional pipe.

2.2.8 ZmqRouterSocket

▷ `ZmqRouterSocket(`*`[uri[, identity]]`*`)` (function)

Router sockets, like dealer sockets, can have multiple peers connected to them. Incoming messages are handled the same way as for dealer sockets. Outgoing messages should be multi-part messages, where the first part of the message is the identity of one of the peers. The message will then be sent only to the peer with that identity. Peers can be dealer, request, or reply sockets.

2.2.9 ZmqSocket

▷ `ZmqSocket(`*`type`*`)` (function)

`ZmqSocket` is a low-level function that is used by `ZmqPushSocket` etc. to create sockets. Its argument is a string, one of “PUSH”, “PULL”, “REP”, “REQ”, “PUB”, “SUB”, “DEALER”, “ROUTER”, and it creates and returns a socket of that type.

2.2.10 ZmqClose

▷ `ZmqClose(`*`socket`*`)` (function)

`ZmqClose` closes `socket`. Afterwards, it cannot anymore be bound or connected to, nor receive or send messages. Messages already in transit will still be delivered.

2.2.11 ZmqIsOpen

▷ `ZmqIsOpen(`*`socket`*`)` (function)

ZmqIsOpen returns true if socket has not been closed yet, false otherwise.

2.2.12 ZmqSocketType

▷ ZmqSocketType(*socket*) (function)

ZmqSocketType returns the string with which the socket was created (see ZmqSocket (2.2.9)).

2.3 Binding and connecting sockets to addresses

2.3.1 ZmqBind

▷ ZmqBind(*socket, uri*) (function)

ZmqBind will *bind* socket to uri. After being bound to the address specified by uri, the socket can be connected to at that address with ZmqConnect (2.3.2).

2.3.2 ZmqConnect

▷ ZmqConnect(*socket, uri*) (function)

ZmqConnect is used to connect socket to another socket that has been bound to uri. Note that you can connect to an address that has not been bound yet; in that case, the connection will be delayed until the binding has occurred.

2.3.3 ZmqAttach

▷ ZmqAttach(*socket, uri*) (function)

ZmqAttach is a unified interface for binding and connecting a socket. If uri begins with a "+" character, then the ZmqConnect (2.3.2) is called with the socket and the rest of the uri string following the "+". Otherwise, ZmqBind (2.3.1) is called with these arguments. The intended use is to construct a network of connections from a list of strings.

2.3.4 ZmqSocketURI

▷ ZmqSocketURI(*socket*) (function)

ZmqSocketURI returns the most recent URI to which socket has been bound or connected. Sockets can be bound to or connected to multiple addresses, but only the most recent one is returned.

2.3.5 ZmqIsBound

▷ ZmqIsBound(*socket*) (function)

ZmqIsBound returns true if the socket has been bound to the address returned by ZmqSocketURI(), false otherwise.

2.3.6 ZmqIsConnected

▷ ZmqIsConnected(*socket*) (function)

ZmqIsBound returns true if the socket has been connected to the address returned by ZmqSocketURI(), false otherwise.

2.4 Sending and receiving messages

ZeroMQ allows the sending and receiving of both string messages and multi-part messages. String messages are sequences of bytes (which can include zero), provided as a GAP string, while multi-part messages are lists of strings, provided as a GAP list. Multi-part messages are largely a convenience feature (e.g., to allow a message to have header parts without the inconvenience of having to encode those in a single string). When sent, multi-part messages will be delivered in their entirety; they can be retrieved one part at a time, but if the first part is available, the last part is available also.

2.4.1 ZmqSend

▷ ZmqSend(*socket, data*) (function)

ZmqSend will send data to socket, according to the routing behavior of the underlying socket mechanism.

2.4.2 ZmqReceive

▷ ZmqReceive(*socket*) (function)

ZmqReceive will either retrieve a string message or a single part of a multi-part message from socket and return the result as a GAP string.

Example

```
gap> z := ZmqSocket("inproc://test");;
gap> z2 := ZmqSocket("+inproc://test");;
gap> ZmqSend(z, "notice");
gap> ZmqReceive(z2);
"notice"
gap> ZmqSend(z, ["alpha", "beta"]);
gap> ZmqReceive(z2);
"alpha"
gap> ZmqReceive(z2);
"beta"
```

2.4.3 ZmqReceiveList

▷ ZmqReceiveList(*socket*) (function)

ZmqReceiveList will retrieve a message in its entirety from socket and return the result as a list of strings.

Example

```
gap> z := ZmqPushSocket("inproc://test");;
gap> z2 := ZmqPullSocket("+inproc://test");;
gap> ZmqSend(z, "notice");
gap> ZmqReceiveList(z2);
[ "notice" ]
gap> ZmqSend(z, ["alpha", "beta"]);
gap> ZmqReceiveList(z2);
[ "alpha", "beta" ]
```

2.4.4 ZmqReceiveListAsString

▷ ZmqReceiveListAsString(*socket*, *separator*) (function)

ZmqReceiveListAsString works like ZmqReceiveList, but will return the result a single string, with multiple parts separated by separator.

Example

```
gap> z := ZmqPushSocket("inproc://test");;
gap> z2 := ZmqPullSocket("+inproc://test");;
gap> ZmqSend(z, "notice");
gap> ZmqReceiveListAsString(z2, "::");
"notice"
gap> ZmqSend(z, ["alpha", "beta"]);
gap> ZmqReceiveListAsString(z2, "::");
"alpha::beta"
```

2.4.5 ZmqHasMore

▷ ZmqHasMore(*socket*) (function)

ZmqHasMore will return true if a socket has one or more remaining parts of a multi-part message outstanding, false otherwise.

Example

```
gap> z := ZmqPushSocket("inproc://test");;
gap> z2 := ZmqPullSocket("+inproc://test");;
gap> ZmqSend(z, "notice");
gap> ZmqReceive(z2);
"notice"
gap> ZmqHasMore(z2);
false
gap> ZmqSend(z, ["alpha", "beta"]);
gap> ZmqReceive(z2);
"alpha"
gap> ZmqHasMore(z2);
true
gap> ZmqReceive(z2);
"beta"
gap> ZmqHasMore(z2);
false
```

2.4.6 ZmqPoll

▷ `ZmqPoll(`*inputs, outputs, timeout*`)` (function)

`ZmqPoll` is a facility to determine if messages can be received from one of the sockets listed in `inputs` or sent to one of the sockets listed in `outputs`. It returns a list of indices describing the sockets that at least one message can be received from or sent to. The timeout is an integer. If positive, it describes a duration (in milliseconds) after which it will return. If zero, the function will return immediately. If it is `-1`, then the function will block indefinitely until at least one message can be retrieved from one of the sockets in `inputs` or at least one message can be sent to one of the sockets in `outputs`. If the timeout is non-negative, the result can be the empty list. It is guaranteed to have at least one element otherwise. The indices in the result are in the range `[1..Length(inputs)+Length(outputs)`. An index `i` less than or equal to `Length(inputs)` refers to the socket `inputs[i]`. An index `j` in the range `[Length(inputs)+1..Length(inputs)+Length(outputs)` refers to the socket `outputs[j-Length(inputs)]`. Multiple indices are listed in ascending order (i.e., they form a GAP set).

Example

```
gap> send1 := ZmqPushSocket("inproc://#1");;
gap> recv1 := ZmqPullSocket("+inproc://#1");;
gap> send2 := ZmqPushSocket();;
gap> recv2 := ZmqPullSocket();;
gap> ZmqSetSendCapacity(send2, 1);
gap> ZmqSetReceiveCapacity(recv2, 1);
gap> ZmqBind(send2, "inproc://#2");
gap> ZmqConnect(recv2, "inproc://#2");
gap> ZmqSend(send2, "alpha");
gap> ZmqSend(send2, "beta");
gap> ZmqPoll([recv1, recv2], [send1, send2], 0);
[ 2, 3 ]
```

In the example above, the code constructs sockets `send2` and `recv2` with a capacity to store at most one outgoing and incoming message, respectively. Then the code sends two messages to `send2`, one of which will be in the incoming buffer of `recv2`, and the other will remain in the outgoing buffer of `send2`. At this point, no more messages can be sent to `send2`, because its outgoing buffer is at capacity, and `recv2` has a message that can be received. Conversely, `send1` can still accept outgoing messages, and `recv1` has no messages. Thus, the result is the list `[2, 3]`. The `2` refers to `recv2` (as the second socket in the list of inputs), while `3` refers to `send1` (as the first socket in the list of outputs).

2.5 Setting and querying socket properties

Sockets have properties that can be set and queried. Most such properties only affect binds and connects that occur after they have been set. Binding or connecting a socket first and then setting a property will not change the behavior of the socket.

2.5.1 ZmqSetIdentity

▷ `ZmqSetIdentity(`*socket, string*`)` (function)

`ZmqSetIdentity` can be used to give the socket an identity. An identity is a string of up to 255 characters that should not start with a null character (the null character is reserved for internal use). This identity should be globally unique. Uniqueness is not enforced, however, and undefined behavior may result from different sockets with the same identity interacting.

2.5.2 ZmqGetIdentity

▷ `ZmqGetIdentity(`*`socket`*`)` (function)

`ZmqGetIdentity` returns the current identity of the socket.

2.5.3 ZmqSetSendCapacity

▷ `ZmqSetSendCapacity(`*`socket, value`*`)` (function)

`ZmqSetSendCapacity` sets the maximum number of messages that a socket can store in its outgoing buffer.

2.5.4 ZmqSetReceiveCapacity

▷ `ZmqSetReceiveCapacity(`*`socket, value`*`)` (function)

`ZmqSetReceiveCapacity` sets the maximum number of messages that a socket can store in its outgoing buffer.

2.5.5 ZmqGetSendCapacity

▷ `ZmqGetSendCapacity(`*`socket`*`)` (function)

`ZmqGetSendCapacity` returns the maximum number of messages that a socket can store in its outgoing buffer.

2.5.6 ZmqGetReceiveCapacity

▷ `ZmqGetReceiveCapacity(`*`socket`*`)` (function)

`ZmqGetReceiveCapacity` returns the maximum number of messages that a socket can store in its incoming buffer.

2.5.7 ZmqSetSendBufferSize

▷ `ZmqSetSendBufferSize(`*`socket, size`*`)` (function)

`ZmqSetSendBufferSize` sets the size of the transmission buffer used by the underlying operating system structure for sending data.

2.5.8 ZmqGetSendBufferSize

▷ `ZmqGetSendBufferSize(`*`socket`*`)` (function)

`ZmqGetSendBufferSize` returns the size of the transmission buffer used by the underlying operating system structure for sending data.

2.5.9 ZmqSetReceiveBufferSize

▷ `ZmqSetReceiveBufferSize(`*`socket, size`*`)` (function)

`ZmqSetReceiveBufferSize` sets the size of the transmission buffer used by the underlying operating system structure for receiving data.

2.5.10 ZmqGetReceiveBufferSize

▷ `ZmqGetReceiveBufferSize(`*`socket`*`)` (function)

`ZmqGetReceiveBufferSize` returns the size of the transmission buffer used by the underlying operating system structure for receiving data.

2.5.11 ZmqSubscribe

▷ `ZmqSubscribe(`*`socket, prefix`*`)` (function)

The `ZmqSubscribe` function can only be used for Subscriber sockets. After calling it, only messages that begin with the given prefix string will be received by the subscriber. All others will be silently discarded. The function can be used multiple times, and then all messages that match any of the prefixes will be received.

2.5.12 ZmqUnsubscribe

▷ `ZmqUnsubscribe(`*`socket, prefix`*`)` (function)

The `ZmqUnsubscribe` function removes the given prefix string from the socket's subscription list.

Chapter 3

Using ZeroMQ with the zgap script

The zgap script provides facilities to start a number of child processes controlled by a single master process and to allow for easy coordination between them.

3.1 Running zgap

From the shell, run zgap via:

Example

```
bin/zgap -N <nodes> <gap_options> <gap_files>
```

Here, nodes should be a positive integer that describes the number of workers one wishes to start. The rest of the command line, consisting of gap options and gap files, will be passed to the master and the worker processes verbatim. This allows, for example, the initialization of functions that need to be known by all workers. The first line of output will be prefixed with [zgap] and will list the directory where zgap will store the files and sockets it uses to communicate. In particular, the logXX.txt files within that directory will contain the output generated by the workers; this is useful for debugging, as the workers do not have a working break loop. Example:

Example

```
bin/zgap -N 4 -P 8 -m 1G common.g
```

On NUMA architectures that support the numactl command, it is possible to further specify which node each worker should be running on. This can take one of two forms:

Example

```
bin/zgap -N <count>:<start>-<end>
bin/zgap -N <count>:+<start>-<end>
```

Each will distribute count worker processes on the physical nodes ranging from start to end in a round-robin fashion, reusing nodes if there are more workers than nodes. The first mode (without a + sign) will use absolute node numbers, the second will be relative to the master process. See the numactl manual page for further details. Example:

Example

```
bin/zgap -N 4:+0-3 -P 8 -m 1G common.g
```

Note: Currently, zgap can only be run from the GAP root directory. This is an implementation restriction that is to be removed at a later date.

3.2 Zgap API

Most of the following API functions take a `dest` argument, which is used to specify the destination of the operation. To specify a worker thread, `dest` would have to be an integer in the range from 1 to the number of worker processes; 0 specifies the master process. Multiple processes can be specified by a range or list of integers. The variable `ZAll` contains a range encompassing the worker processes; `ZSelf` contains the index of the current worker or 0 for the master.

3.2.1 ZExec

▷ `ZExec(`*`dest, cmd`*`)` (function)

This function sends `cmd` to the given destination and executes it there. The command must be a valid GAP statement ending in a semicolon. If `dest` specifies multiple processes, the command will be executed on all of them.

3.2.2 ZBind

▷ `ZBind(`*`dest, var, expr`*`)` (function)

This function binds the global variable described by the string `var` to the value `expr` in all processes listed in `dest`. Note that `expr` must evaluate to a serializable value.

Example

```
gap> ZBind(ZAll, "counter", 0);
```

3.2.3 ZUnbind

▷ `ZUnbind(`*`dest, var`*`)` (function)

This function is the counterpart to `ZBind`. It will unbind `var` in all specified processes.

Example

```
gap> ZUnbind(ZAll, "status");
```

3.2.4 ZCall

▷ `ZCall(`*`dest, func, args`*`)` (function)

This function will execute the function specified by the string `func` in the specified processes. The string `func` must be the name of a global variable referring to the function to be executed. This function should be created at startup by adding a file to the commandline that defines it in all workers or by `ZExec`.

Example

```
gap> ZBind(ZAll, "counter", 0);
gap> ZExec(Zall, "add := function(n) counter := counter + n; end;");
gap> ZCall(1, "add", [1]);
```

3.2.5 ZQuery

▷ ZQuery(*dest, func, args, callback*) (function)

This function works like ZCall, except that any return value will be passed to the callback function.

Example

```
gap> res := false;
false
gap> ZQuery(1, "ReturnTrue", [], function(x) res := x; end);
gap> res;
true
```

3.2.6 ZResponse

▷ ZResponse() (function)

ZResponse is a convenience function to construct blocking callbacks for ZCall and ZTask. It returns a record containing a put, a get, and a test function. Here, put is passed as the callback; get can be used to read the returned value; and test can be used to test for the presence of a value.

Example

```
gap> resp := ZResponse();;
gap> ZQuery(1, "Z", [4], resp.put);
gap> resp.get();
Z(2^2)
gap> resp.test();
true
```

3.2.7 ZTask

▷ ZTask(*dest, func, args, callback*) (function)

This function works like ZQuery, except that the function will be executed via a task and callback will be called after the task finishes and returns a result.

3.2.8 ZAsync

▷ ZAsync(*dest, func, args*) (function)

This function works like ZCall, except that the function will be executed via a task.

3.2.9 ZRead

▷ ZRead(*dest, file*) (function)

This function does a Read(file) for all specified processes.

3.2.10 ZReadGapRoot

▷ `ZReadGapRoot(`*`dest, file`*`)` (function)

This function does a `ReadGapRoot(file)` for all specified processes.

Index

ZAsync, 16
ZBind, 15
ZCall, 15
ZExec, 15
ZmqAttach, 8
ZmqBind, 8
ZmqClose, 7
ZmqConnect, 8
ZmqDealerSocket, 7
ZmqGetIdentity, 12
ZmqGetReceiveBufferSize, 13
ZmqGetReceiveCapacity, 12
ZmqGetSendBufferSize, 13
ZmqGetSendCapacity, 12
ZmqHasMore, 10
ZmqIsBound, 8
ZmqIsConnected, 9
ZmqIsOpen, 7
ZmqPoll, 11
ZmqPublisherSocket, 6
ZmqPullSocket, 6
ZmqPushSocket, 6
ZmqReceive, 9
ZmqReceiveList, 9
ZmqReceiveListAsString, 10
ZmqReplySocket, 6
ZmqRequestSocket, 6
ZmqRouterSocket, 7
ZmqSend, 9
ZmqSetIdentity, 11
ZmqSetReceiveBufferSize, 13
ZmqSetReceiveCapacity, 12
ZmqSetSendBufferSize, 12
ZmqSetSendCapacity, 12
ZmqSocket, 7
ZmqSocketType, 8
ZmqSocketURI, 8
ZmqSubscribe, 13
ZmqSubscriberSocket, 7
ZmqUnsubscribe, 13
ZQuery, 16
ZRead, 16
ZReadGapRoot, 17
ZResponse, 16
ZTask, 16
ZUnbind, 15