

YAMI Core Library Reference

Copyright © 2001-2008 Maciej Sobczak

Contents

1	Introduction	2
2	Parameter Set	3
3	Agent	7
3.1	Helper definitions	7
3.2	Policies	7
3.3	General API	9
3.4	Domains management	10
3.5	Objects management	10
3.6	Sending new messages	12
3.7	Incoming messages	15
3.8	Additional functions	18
4	Synchronization Primitives	20
5	Error Codes	22
5.1	Logic error codes	22
5.2	Run-time error codes	24
6	YAMI Constants	26
6.1	Specification constants	26
6.2	Default values	26
6.3	Implementation choices	27
7	Compilation options and possible problems	28
7.1	Compilation for non-threaded platforms	28

1 Introduction

The *YAMI Core Library* consists of two different parts:

- the part that is supposed to be used directly by the end-user (by the programmer who wants to write programs using *YAMI*):
 - Parameter Set API
 - Agent API
 - Synchronization Primitives
 - Error Codes
- the part that is not supposed to be used by the client code, but it can be useful for programmers writing their own low-level code:
 - Domain Set
 - Queue
 - XDR Module
 - Operating System Layer

Please refer to the comments in source code for further information about this second part.

To be exact, *Synchronization Primitives* are part of *Operating System Layer* that were made available to normal users to help them solve common problems in multithreaded *YAMI* applications.

Note: *The YAMI library is not intended to be a library for multithreaded programming in general, but it can be freely mixed with other libraries providing their own primitives.*

Every *YAMI* function returns `YAMIOK` integer code in case of success, unless otherwise specified.

2 Parameter Set

Parameter Set is the object used by the client code to encapsulate the functionality of the variable length sequence of variable type parameters. This object is not internally synchronized and is intended for use in one thread and per one message (however, the latter constraint is not strict).

Parameter Set is created when there is a need to send some data to the remote object and conceptually is similar to the set of parameters of the function or method.

In order to use *Parameter Set* functionality, the user code has to include `yamiparams.h` header.

```
typedef void* HPARAMSET;
```

Type of the handle to a *Parameter Set*.
The handle is *opaque* to the client.

```
enum paramType
{
    eNoType,
    eString,
    eWString,
    eInt,
    eDouble,
    eByte,
    eBinary
};
```

Enums defining types used in *Parameter Set*.
`eNoType` is used internally by the implementation.

```
int yamiCreateParamSet(HPARAMSET *pps, int parcount);
```

Creates a *Parameter Set* of the length designated by `parcount`, no parameter is initialized.

Typical error codes: `YAMIOUTOFRANGE` (this may happen when `parcount` is invalid), `YAMINOMEMORY`.

```
int yamiDestroyParamSet(HPARAMSET paramset);
```

Destroys a *Parameter Set*.

Typical error codes: `YAMIBADHANDLE`.

```
int yamiCopyParamSet(HPARAMSET *ppsdst, HPARAMSET pssrc);
```

Creates a copy of the *Parameter Set*.

The `pssrc` parameter is a handle to the existing (source) object and the `ppdst` parameter is a pointer to the place, where the handle of the copy (destination) will be stored.

Typical error codes: YAMIBADHANDLE, YAMINOMEMORY.

```
int yamiCopyParameter(HPARAMSET psdst, int positiondst,
                    HPARAMSET pssrc, int positionsrc);
```

Copies a parameter from one *Parameter Set* to the other.

Typical error codes: YAMIBADHANDLE, YAMINOMEMORY.

```
int yamiClearParameter(HPARAMSET paramset, int position);
```

Clears the given entry.

Typical error codes: YAMIBADHANDLE, YAMIOUOFRANGE.

```
int yamiGetParamCount(HPARAMSET paramset, int *pparcount);
```

Gets the number of parameters in the *Parameter Set*.

Typical error codes: YAMIBADHANDLE.

```
int yamiGetType(HPARAMSET paramset, int position,
                enum paramType *ptype);
```

Gets the type of the parameter at a given position.

Typical error codes: YAMIBADHANDLE, YAMIOUOFRANGE.

```
int yamiSetInt          (HPARAMSET paramset, int position,
                        int ivalue);
int yamiSetByte        (HPARAMSET paramset, int position,
                        char cvalue);
int yamiSetDouble      (HPARAMSET paramset, int position,
                        double dvalue);
int yamiSetBinary      (HPARAMSET paramset, int position,
                        const char *pbuf, int bufsize);
int yamiSetBinaryShallow (HPARAMSET paramset, int position,
                        const char *pbuf, int bufsize);
int yamiSetString      (HPARAMSET paramset, int position,
                        const char *pstr);
int yamiSetStringShallow (HPARAMSET paramset, int position,
                        const char *pstr);
int yamiSetWString     (HPARAMSET paramset, int position,
                        const wchar_t *pwstr);
int yamiSetWStringShallow (HPARAMSET paramset, int position,
                        const wchar_t *pwstr);
```

Set a value of appropriate type in a *Parameter Set* at a given position.

The `yamiSetXXXShallow` functions allow to set a parameter value *without* copying the data – only the pointers to the given buffer are set up in the *Parameter Set*. The user should not make any changes to the buffers used in this way through the lifetime of the *Parameter Set* object.

Typical error codes: YAMIBADHANDLE, YAMINOMEMORY, YAMIOUOFRANGE.

```
int yamiGetInt    (HPARAMSET paramset, int position,
                  int *pivalue);
int yamiGetByte  (HPARAMSET paramset, int position,
                  char *pcvalue);
int yamiGetDouble(HPARAMSET paramset, int position,
                  double *pdvalue);
```

Retrieve the value from the *Parameter Set*, from the specified position.

Typical error codes: YAMIBADHANDLE, YAMIOUOFRANGE, YAMIBADTYPE.

```
int yamiGetBinarySize (HPARAMSET paramset, int position,
                       int *psize);
int yamiGetBinaryValue (HPARAMSET paramset, int position,
                        char *pbuf);
int yamiGetBinaryBuffer(HPARAMSET paramset, int position,
                        const char **ppbuf);
```

Retrieve the size of the binary data in a parameter and the data itself. The buffer of the appropriate size should be allocated by the client calling the `yamiGetBinaryValue` function.

Typical error codes: YAMIBADHANDLE, YAMIOUOFRANGE, YAMIBADTYPE.

```
int yamiIsASCIIString(HPARAMSET paramset, int position,
                      int *isascii);
```

Checks if the parameter is a plain, narrow ASCII string. If yes, `*isascii` will have value 1 or 0 otherwise.

```
int yamiGetStringLength(HPARAMSET paramset, int position,
                         int *plength);
int yamiGetStringValue (HPARAMSET paramset, int position,
                        char *pbuf);
int yamiGetStringBuffer(HPARAMSET paramset, int position,
                        const char **ppbuf);
```

Retrieve the length (number of characters, without terminating zero) of the string in a parameter and the string itself. The buffer of the appropriate size (length + place for terminating zero) should be allocated by the client calling the `yamiGetStringValue` function.

Typical error codes: YAMIBADHANDLE, YAMIOUTOFRANGE, YAMIBADTYPE.

```
int yamiGetWStringLength(HPARAMSET paramset, int position,
                        int *plength);
int yamiGetWStringValue (HPARAMSET paramset, int position,
                        wchar_t *pbuf);
int yamiGetWStringBuffer(HPARAMSET paramset, int position,
                        const wchar_t **ppbuf);
```

Retrieve the length (number of characters, without terminating zero) of the wide string in a *Parameter Set* and the string itself. The buffer of the appropriate size (length + place for terminating zero) should be allocated by the client calling the `yamiGetWStringValue` function.

Typical error codes: YAMIBADHANDLE, YAMIOUTOFRANGE, YAMIBADTYPE.

Note:

It is perfectly possible to set a value as a **String** and retrieve it as a **WString**. It is possible to set a value as a **WString** and retrieve it as a **String** only if the value set does not contain characters outside of the ASCII range. In both cases, the *Parameter Set* object performs the conversion.

```
int yamiGetPackedSize      (HPARAMSET paramset, int *packedsize);
int yamiGetPacked          (HPARAMSET paramset, char *pbuf);
int yamiCreatePSFromPacked(HPARAMSET *pps, int levelno,
                          const char *pbuf, int swap);
```

Pack the *Parameter Set* into the sequence of bytes that is appropriate for further transmission (the so-called *marshalling*) and create the *Parameter Set* from the sequence of bytes.

The buffer of the appropriate size should be allocated by the client calling this function.

The `levelno` parameter declares the level of *YAMI* specification. If the byte sequence doesn't conform to this level, the `YAMIPARSEERROR` will be reported.

If the `swap` parameter is non-zero, the bytes representing multi-byte data (like integer values) will be swapped to the reverse order.

These functions are not supposed to be called by the client code, although users can achieve some *persistence* functionality with them.

Typical error codes: YAMIBADHANDLE, YAMINOTINITIALIZED, YAMINOMEMORY, YAMIPARSEERROR.

```
int yamiCheckRawBuffer(int levelno, const char *pbuf, int swap);
```

Checks if the buffer contains a valid data for a *Parameter Set*.

The `levelno` parameter declares the level of *YAMI* specification.

This function is not supposed to be called by the client code.

Typical error codes: YAMIPARSEERROR.

3 Agent

Agent is responsible for routing, sending and receiving messages over the network. Conceptually, it is similar to the ORB in CORBA systems.

The *Agent* object is created in (almost) every program that uses the *YAMI* library, no matter if the given component acts as a *client* or as a *server* in the distributed system.

The multi-threading abilities (and other things that can vary between different *Agents*) of the *Agent* object greatly depend on the parameters that are provided during its creation – these parameters are called *Policies*.

In order to use *Agent* functionality, the user code has to include `yamic.h` header.

3.1 Helper definitions

```
enum eAgentLevel
{
    eLevel1 = 1,
    eLevel2 = 2
};
```

Enums defining levels of typing, for better verbosity where the level parameter is required.

3.2 Policies

```
struct yamipolicies
{
    int agentlevel; /* 2 */
    int objqmaxlength; /* 10 */
    int objqmaxsize; /* 1048576 */
    int dispatchers; /* 1 */
    int senders; /* 1 */
    int sqmaxlength; /* 256 */
    int sqmaxsize; /* 1048576 */
    int connpoolsize; /* 10 */
    int sendtries; /* 5 */
    int hassocket; /* 1 */
    int reuseaddr; /* 1 */
    int hasreceiver; /* 1 */
    int haswaker; /* 1 */
    int mtuser; /* 1 */
    int allowduplex; /* 1 */
    int maxdplxconns; /* 100 */
    int receiveridle; /* 500 */
};
```

Set of *Policies* that can be passed during the *Agent's* creation. Meaning of the *Policies* (the default values are shown in comments above):

agentlevel The level of the *YAMI* specification that this *Agent* will conform to. The *Agent* will not send nor receive messages from incompatible levels.

objmaxlength The maximum length of the queue that is kept for each registered object. If the queue is full with respect to this parameter and there is new incoming message for the given object, it will be rejected with `eOverflow` notification sent to the remote site (to the *Agent* that sent the message).

objmaxsize The maximum size of the same queue (in bytes).

dispatchers The number of threads that the *Agent* will use for message dispatching. If set to 1, all the calls to servants will be serialized, even if there are registered many different passive objects. If this parameter is more than 1, the user code has to assure that the servants are safe with respect to multiple threads. If set to 0, the calls to passive objects are made directly by the receiver module.

senders The number of threads used for sending. If set to non-zero value, the sender thread(s) take care of all the packets that need to be sent to the remote *Agents*. If set to 0, no sender thread is created and the sending is performed directly in the context of requesting thread.

sqmaxlength The maximum length of queue for sender thread.

sqmaxsize The maximum size of the same queue.

connpoolsize The size of the connection pool (applies both to sending and receiving connections).

sendtries The number of times the sender module should try to send the packet before giving up (and reporting an error), if there is a network problem.

hassocket The flag stating if the *Agent* should create the listening socket. If set to zero, no receiving will be possible (this option can be useful for lightweight *Agents* that are created for one-way messaging).

reuseaddr The flag stating if the *Agent's* listening socket should be created with the `SO_REUSEADDR` option (when the flag has non-zero value). This policy can be useful for *Agents* that are frequently created and destroyed on the same port.

hasreceiver The flag stating if the separate receiver thread is needed. If set to non-zero value, the **hassocket** flag should be non-zero, too.

haswaker The flag stating if the waker module is needed. The waker module requires a separate thread for its own and in some lightweight configurations is not necessary.

mtuser The flag stating if the user code is multi-threaded. If the value is non-zero, the internal synchronization is provided so that all the *Agent*'s functions are thread-safe. In single-threaded code, setting this value to 0 causes the *Agent* to bypass the internal synchronization whenever possible.

allowduplex The flag stating if the *Agent* should accept incoming duplex connections. Accepting duplex connections has influence on the amount of network resources that can be potentially consumed by the server *Agent*.

maxdplxconns The number of incoming duplex connections that are kept in a pool. If the pool is full with respect to this parameter, no new incoming duplex connection will be accepted.

receiveridle The maximum idle time for the receiver thread, in milliseconds. This time is the maximum delay between sending the duplex request and including the socket in the set checked for incoming packets (when this is the first duplex request to the given address). Note that very short idle times may implicate more CPU resources used by the receiver thread.

```
void yamiGetDefaultPolicies(struct yamipolicies *ppolicies);
```

Fills the *Policies* structure with the default values.

3.3 General API

```
typedef void* HYAMIAGENT;
```

Type of the handle to an *Agent* object.
The handle is *opaque* to the client.

```
int yamiCreateAgent(HYAMIAGENT *pagent, int port,
                   struct yamipolicies *ppolicies);
```

Creates a new *Agent* (and thus - new domain), working on the given IP port and with the given set of *Policies*.

If the `port` is 0, the *Agent* starts with the port number assigned by the operating system. If the `ppolicies` is NULL, the default values are used.

Typical error codes: YAMIOUOFRANGE, YAMINETEROR.

```
int yamiDestroyAgent(HYAMIAGENT agent);
```

Destroys the *Agent* and all its internal structures.

Typical error codes: YAMIBADHANDLE.

3.4 Domains management

```
enum connectionOptions
{
    eNone          = 0,
    eFixedDuplex  = 1
};
```

Enums defining connection options, differentiating between simplex (no connection option) and duplex mode.

```
int yamiAgentDomainRegister(HYAMIAGENT agent,
                           const char *name, const char *addr,
                           int port, int level);
int yamiAgentDomainRegisterEx(HYAMIAGENT agent,
                              const char *name, const char *addr,
                              int port, int level, int options);
int yamiAgentDomainUnregister(HYAMIAGENT agent,
                              const char *name);
```

Registers and unregisters a domain in the *Agent*'s private domain set.

name will identify the new entry in the address book.

addr is the network address; it can have a "xxx.xxx.xxx.xxx" form (for example "10.1.12.13") or a human-readable form (for example "comp.company.com").

port is a port number of the destination *Agent*.

level is a level understandable by the destination *Agent*.

options is one of the options defined in the `connectionOptions` enum type. Note that this parameter is ignored when the message is *forwarded* to the given domain – forwards are always performed in the simplex mode.

Typical error codes: YAMIBADHANDLE,
YAMICANNOTRESOLVEADDRESS, YAMINOTFOUND.

3.5 Objects management

```
enum objectType
{
    polling,
    passive_singlethreaded,
    passive_multithreaded
};
```

Types of objects registered in the *Agent*.

polling The object is responsible for asking the *Agent* for incoming messages.

passive_singlethreaded The object will be *up-called* for each incoming message, but no more than one thread at a time will perform up-calls in case of many messages waiting in a queue.

passive_multithreaded Like the previous, but possibly more than one of the *Agent*'s threads can perform up-calls when many messages arrive.

```
typedef void (*DISPFUNCTION)(HINCMMSG inc);
```

Type of the *dispatch function*, which will be called for a given object (if registered as *passive*) when the incoming message arrives.

The message will be considered *rejected*, unless the dispatch function has processed the message in some way. It means, that if the *dispatch function* did not process the message, it will be automatically rejected and the client (calling object) will receive appropriate notification.

The incoming message structure (handled by *inc*) is *owned* by the *Agent*, so the dispatch function *should not* release it by itself.

See later for description of HINCMMSG handle.

```
#define YAMI_ANY_OBJECT NULL
```

The generic name which allows to register objects that will accept messages, when no other object name can match the object name in the message sent. This facility is provided for implementing *default* objects.

```
int yamiAgentObjectRegister(HYAMIAGENT agent, const char *name,
                           enum objectType otype,
                           DISPFUNCTION df, void *hint);
```

Registers new object. The registration includes creation of the incoming message queue, which means that incoming messages are accepted and queued immediately after this function completes.

name is the object's name or YAMI_ANY_OBJECT. Only messages that have the destination object name matching with this parameter will be queued.

If the type is *polling*, then the *df* parameter is ignored. For *passive_{}*... objects, it's the address of the function to call for each incoming message.

hint is a parameter that can be retrieved with each incoming message using *yamiAgentIncomingMsgGetHint* function.

Typical error codes: YAMIBADHANDLE.

```
int yamiAgentObjectUnregister(HYAMIAGENT agent,
                             const char *name);
```

Unregisters the object. This means, that all waiting messages for this object (and all new that will be received) will be discarded with the *unknown object* reply message.

Typical error codes: YAMIBADHANDLE, YAMINOTFOUND.

3.6 Sending new messages

```
typedef void* HMESSAGE;
```

Type of the handle to a message object.
The handle is *opaque* to the client.

```
enum msgStatus
{
    ePosted,
    ePending,
    eReplied,
    eRejected,
    eUnknownObj,
    eOverflow,
    eNetError,
    eTimedOut,
    eRejectByAgent
};
```

The states that the message sent can be in.

ePosted The message was posted to the queue owned by the sender thread.

ePending The message was successfully sent over the network.

eReplied There is a reply available for the given message.

eRejected The message was rejected by the remote object (but was received).

eUnknownObj The message was rejected by the remote *Agent*, because no object was registered with a matching name.

eOverflow The message was rejected by the remote *Agent*, because of the overflow in the remote object's queue.

eNetError The message was not successfully sent over the network.

eTimedOut The message was *timed out* by the waker module.

eRejectByAgent The message was rejected by the destination *Agent*. This may happen if the message with the parameter set of Level2 was sent to the *Agent* running on Level1.

```
#define YAMI_THIS_DOMAIN NULL
```

This is the name that always corresponds to the same *Agent* through which the message is being sent.

This facility is provided for setting up the communication between objects living in the same domain.

Registering the domain with the same address and IP port as the *Agent* and sending the message using the name of that domain also works. In both cases, the *shortcut routing* will be performed.

```
int yamiAgentMsgSend(HYAMIAGENT agent, const char *domainname,
                    const char *objectname,
                    const char *messagename,
                    HPARAMSET paramset, HMESSAGE *phm);
```

Sends new message.

domainname is a name of the registered domain, where the message will be sent. It can be `YAMI_THIS_DOMAIN`.

objectname is a name of the remote object.

messagename is a name of the message being sent.

paramset is a handle to the *Parameter Set* that will be sent together with the message.

phm will receive the handle to the message structure. The client code can use this handle for retrieving the message status, the replies, and so on. If this parameter is `NULL`, no message structure is created and the message is of type *one-way*.

Typical error codes: `YAMIBADHANDLE`.

```
int yamiAgentMsgSendAddr(HYAMIAGENT agent,
                        const char *addr, int port, int level,
                        const char *objectname,
                        const char *messagename,
                        HPARAMSET paramset, HMESSAGE *phm);
int yamiAgentMsgSendAddrEx(HYAMIAGENT agent,
                           const char *addr, int port, int level,
                           const char *objectname,
                           const char *messagename,
                           HPARAMSET paramset, HMESSAGE *phm,
                           int options);
```

Sends new message to explicitly given address.

addr is the network address; it can have a "xxx.xxx.xxx.xxx" form (for example "10.1.12.13") or a human-readable form (for example "comp.company.com").

port is a port number of the destination *Agent*.

level is a level understandable by the destination *Agent*. *objectname* is a name of the remote object.

messagename is a name of the message being sent.

paramset is a handle to the *Parameter Set* that will be sent together with the message.

phm will receive the handle to the message structure. The client code can use this handle for retrieving the message status, the replies, and so on. If this

parameter is NULL, no message structure is created and the message is of type *one-way*.

`options` is one of the options defined in the `connectionOptions` enum type.

Typical error codes: YAMIBADHANDLE.

```
int yamiAgentSetUpTimeout(HMESSAGE message, int timeout);
```

Sets the timeout for a given message.

`timeout` holds the number of seconds, after which the waker module will set the status of the message to `eTimedOut` (provided that the status of the message was still in the `ePosted` or `ePending` state).

This function allows to write recovery code for clients that without it would wait forever for replies.

This function does not have to be used (but can) when the client periodically polls for the message's status and can decide by itself that the timeout expired, but can be of great help for clients that decide to *wait* until the reply arrives.

Typical error codes: YAMIBADHANDLE, YAMIOUTOFRANGE.

```
int yamiAgentMsgGetStatus(HMESSAGE hm, enum msgStatus *status);
```

Retrieves the message's status.

Typical error codes: YAMIBADHANDLE.

```
int yamiAgentMsgWait(HMESSAGE hm);
```

This function *blocks* the calling thread until the status of the message changes to one of the codes: `eReplied`, `eRejected`, `eUnknownObj`, `eOverflow`, `eNetError`, `eTimedOut` or `eRejectByAgent`.

When this happens for any reason, the calling thread wakes up and should check for the message's status.

Clients calling this function should consider also `yamiAgentSetUpTimeout`, because without it, the `yamiAgentMsgWait` can block *forever* (which can happen when the remote object crashes in the middle of processing).

Typical error codes: YAMIBADHANDLE.

```
int yamiAgentMsgGetResponse(HMESSAGE hm, HPARAMSET *pps);
```

This function retrieves the *Parameter Set* that was sent back by the remote object.

Typical error codes: YAMIBADHANDLE, YAMINOTAVAILABLE.

```
int yamiAgentMsgDestroy(HMESSAGE hm);
```

Destroys the message structure.

Typical error codes: YAMIBADHANDLE.

Retrieves the *Parameter Set* that was sent together with the message. This function can return YAMINOTAVAILABLE if the message was sent without parameters.

Typical error codes: YAMIBADHANDLE, YAMINOTAVAILABLE.

```
int yamiAgentIncomingMsgGetLevel(HINCMMSG inc, int *levelno);
```

Retrieves the level of *YAMI* specification, to which the sending object complies. It can help to choose the best reply, according to the abilities of the remote (sending) object.

Typical error codes: YAMIBADHANDLE.

```
int yamiAgentIncomingMsgGetHint(HINCMMSG inc, void **phint);
```

Retrieves the *hint* parameter that was set during the object's registration.

Typical error codes: YAMIBADHANDLE.

```
int yamiAgentIncomingMsgGetSourceAddr(HINCMMSG inc, char *pbuf);
```

Writes in the provided buffer the network address of the sending *Agent* in the standard form *xxx.xxx.xxx.xxx*. This address is preserved with message forwarding, which means that it relates to the *Agent* that originated the message, not to the one that forwarded it.

Typical error codes: YAMIBADHANDLE.

```
int yamiAgentIncomingMsgGetSourcePort(HINCMMSG inc, int *pport);
```

Retrieves the port number of the sending *Agent*, see above for the exact meaning.

Typical error codes: YAMIBADHANDLE.

```
int yamiAgentIncomingMsgGetEphemAddr(HINCMMSG inc, char *pbuf);
```

Writes in the provided buffer the network address of the connecting (remote) socket, in the standard form *xxx.xxx.xxx.xxx*. This address might be different than the source address in case of forwarded message. The intent of this function is to allow programs to send outgoing messages to remote agents via duplex channels initiated already by those remote agents.

Typical error codes: YAMIBADHANDLE.

```
int yamiAgentIncomingMsgGetEphemPort(HINCMSG inc, int *pport);
```

Retrieves the (ephemeric) port number of the connecting socket, see above for details and intent.

Typical error codes: YAMIBADHANDLE.

```
int yamiAgentIncomingMsgEat(HINCMSG inc);
```

Marks the incoming message as already processed. The intent of this is to *sink* the message in the *Agent* so that no response (neither reply nor reject) is possible. This function is mainly needed for *passive* objects for implementing the *one-way* message semantics (so that the message is sent and received but no return information is sent to the originating object).

Typical error codes: YAMIBADHANDLE.

```
int yamiAgentIncomingMsgReject(HINCMSG inc);
```

Marks the incoming message as rejected. No further processing (for example sending reply) will be possible.

Typical error codes: YAMIBADHANDLE.

```
int yamiAgentIncomingMsgReply(HINCMSG incoming,
                               HPARAMSET paramset);
```

Sends the reply, together with the *Parameter Set* to the remote object.

Typical error codes: YAMIBADHANDLE.

```
int yamiAgentIncomingMsgForward(HINCMSG inc,
                                const char *domainname,
                                const char *objectname,
                                const char *messagename,
                                HPARAMSET paramset);
```

Forwards the incoming message to the object designated by *objectname* to the domain registered as *domainname*.

It is possible to change the name of the message and its *Parameter Set*. This can be useful for load-balancing, routers or adapters.

Typical error codes: YAMIBADHANDLE.

```
int yamiAgentIncomingMsgForwardAddr(HINCMMSG inc,
                                     const char *domainaddr,
                                     int domainport,
                                     int domainlevel,
                                     const char *objectname,
                                     const char *messagename,
                                     HPARAMSET paramset);
```

Forwards the incoming message to the object designated by `objectname` to the domain with explicitly given address.

`domainaddr` is the network address of the destination domain; it can have a "xxx.xxx.xxx.xxx" form (for example "10.1.12.13") or a human-readable form (for example "comp.company.com").

`domainport` is a port number of the destination *Agent*.

`domainlevel` is a level understandable by the destination *Agent*. `objectname` is a name of the remote object.

`messagename` is a name of the message being sent.

`paramset` is a handle to the *Parameter Set* that will be sent together with the message.

It is possible to change the name of the message and its *Parameter Set*. This can be useful for load-balancing, routers or adapters.

Typical error codes: YAMIBADHANDLE.

```
int yamiAgentDestroyIncomingMsg(HINCMMSG inc);
```

Destroys the incoming message structure.

Typical error codes: YAMIBADHANDLE.

3.8 Additional functions

```
int yamiEasySend(const char *address, int port, int level,
                 const char *objectname, const char *messagename,
                 HPARAMSET paramset);
```

Creates an *ad-hoc* lightweight *Agent* and sends a *one-way* message to the given domain and object.

The *Agent* created for sending the message is immediately destroyed, so this function is provided for convenience where messages are sent rarely or where the performance of communication infrastructure is of no concern. For purposes where many messages are sent to remote objects, it is preferred to create full *Agent* object once and use it for sending many messages.

Typical error codes: YAMIBADHANDLE, YAMINETERROR.

```
int yamiAgentGetLocalAddress(HYAMIAGENT agent, char *pbuf);
```

Writes in the provided buffer the local network address in the standard form *xxx.xxx.xxx.xxx*.

It can be useful for objects that need to pass their location to remote objects, but see also `yamiAgentIncomingMsgGetSourceAddr`.

Typical error codes: YAMIBADHANDLE.

```
int yamiAgentGetLocalPort(HYAMIAGENT agent, int *pport);
```

Retrieves the port number that is actually used by the listening socket.

If the *Agent* was created without the listening socket, this function returns YAMINOSUCHSERVICE.

It can be useful for user code that created *Agent* with the default port.

Typical error codes: YAMIBADHANDLE, YAMINOSUCHSERVICE.

```
int yamiAgentProcessEvent(HYAMIAGENT agent, int timeout);
```

Processes the single event on the listening socket or on the receiving connection pool. The function returns after the `timeout` milliseconds with the YAMINOTAVAILABLE code if there is no incoming connection or immediately with YAMINOTFOUND if there is no socket to wait for. The negative value of the `timeout` causes the function to block until there is some incoming packet to process. The incoming packet is processed in the same way as by the receiver thread – in fact, this function implements one iteration of the receiving module. It is provided for those who want to have complete control everywhere.

This function requires that the *Agent* has a listening socket, but no receiver thread (see policies).

Note: *In explicit event loop, the responsiveness of the Agent (as seen by other communicating components) depends on the frequency of event processing. This function is supposed to be used in tight loops.*

Typical error codes: YAMIBADHANDLE, YAMINOSUCHSERVICE, YAMINOTFOUND, YAMINOTAVAILABLE.

```
int yamiNetInitialize(void);
int yamiNetCleanup(void);
```

Perform the network initialization and cleanup on platforms which require this (for example Windows).

They are empty functions otherwise, but programmers should use them anyway as a mean of writing portable code.

These functions can be omitted if the client code does perform required initialization by itself (because it uses the network for other purposes).

Typical error codes: YAMINETERROR.

4 Synchronization Primitives

The synchronization primitives are provided for convenience and to help resolve problems that naturally arise in multi-threading environments. In *YAMI*, multi-threading can be a consequence of using *passive* objects, especially in combination with multiple dispatching threads. Users of the *YAMI* library can use any set of synchronization primitives from any other library (including the native primitives available on the specific platform). In this context, the synchronization primitives in the *YAMI* core library are provided only for completeness.

In order to use synchronization functionality, the user code has to include `yamisynchro.h` header.

```
typedef void* HMUTEX;
typedef void* HSEMAPHORE;
```

Types of the handles to mutex and semaphore objects.
The handles are *opaque* to the client.

```
int yamiCreateMutex(HMUTEX *pm);
int yamiDestroyMutex(HMUTEX m);
```

Creates and destroys the mutex object.

Typical error codes: YAMISYNCHROERROR.

```
int yamiMutexLock(HMUTEX m);
int yamiMutexUnlock(HMUTEX m);
```

Locks and unlocks the mutex object.

Typical error codes: YAMIBADHANDLE, YAMISYNCHROERROR.

```
int yamiCreateSemaphore(HSEMAPHORE *ps, int initval);
int yamiDestroySemaphore(HSEMAPHORE s);
```

Creates and destroys the semaphore object.
`initval` is the initial value of the semaphore object.

Typical error codes: YAMISYNCHROERROR.

```
int yamiSemaphoreAcquire(HSEMAPHORE s);
int yamiSemaphoreTryAcquire(HSEMAPHORE s);
int yamiSemaphoreRelease(HSEMAPHORE s);
```

Standard *p* and *v* operations on the semaphore object.
If the semaphore is not available, the `yamiSemaphoreTryAcquire` function returns `YAMINOTAVAILABLE`.

Typical error codes: YAMIBADHANDLE, YAMISYNCHROERROR,
YAMINOTAVAILABLE.

```
int yamiSleep(int timeout);
```

Stops the calling thread for `timeout` milliseconds. The value 0 causes the calling thread to stop for infinite period of time.

Typical error codes: YAMISYNCHROERROR.

5 Error Codes

With very small exceptions, all the *YAMI* core library functions return some error code.

All error codes are preprocessor macros with integer values and are divided into two groups:

- *logic errors* – these are programming errors that result from wrong usage of the *YAMI* library
- *run-time errors* – these are errors that result from factors that are outside of the *YAMI* library, for example network problems, memory shortages, etc.

In order to use the *YAMI* error codes, the user code needs to include `yamierrors.h` header.

```
const char * yamiErrorString(int cc);
```

Returns a human-readable description of a given error code.
If the code is not defined, returns `NULL` pointer.
This function can help in debugging.

YAMIOK

No error. The operation completed successfully.

5.1 Logic error codes

YAMIISLOGICERROR(e)

The macro that resolves to 1 if the provided error code is in the logic errors group and 0 otherwise.

YAMIBADHANDLE

This is returned when the handle passed is invalid (has a `NULL` value) or points to the object of unexpected type.

YAMIOUOFRANGE

This is returned when the argument passed is out of the valid range, for example when referring to the parameter position in a *Parameter Set*.

YAMICANNOTINITIALIZETWICE

This error code is no longer used.

YAMIBADTYPE

This is returned when there is a request for a parameter in a *Parameter Set* that has incompatible type. For example, the parameter holds the integer value and the double was requested.

YAMINOTINITIALIZED

This is returned while trying to pack the *Parameter Set* where not all parameters were already initialized or when the user tries to retrieve the parameter value that was not yet set.

YAMIALREADYREGISTERED

This is returned while trying to register a domain or object with the name that was already registered.

YAMINOTFOUND

This is returned when looking up unknown entity (for example domain or object that was not registered).

YAMIQUEUEEMPTY

This is returned when trying to pop a packet while the queue is empty. This error is used internally and should never appear in the user code.

YAMIPARSEERROR

This is returned when trying to parse an invalid packet.

YAMICANNOTPROCESSTWICE

This is returned while trying to reply a second time to the same incoming message or to reply when the message was already rejected, eaten, etc.

YAMIBADLENGTH

This is returned when the user code provides parameters (especially string or binary values), which lengths are outside of the *YAMI* specification.

YAMINOSUCHSERVICE

This is returned when the client code tries to use a feature that was not provided. For example, when the user creates an *Agent* object without the waker module and later tries to set up time-outs on some message.

5.2 Run-time error codes

YAMIISRUNTIMEERROR(*e*)

The macro that resolves to 1 if the provided error code is in the run-time errors group and 0 otherwise.

YAMINOTAVAILABLE

This can be returned when a resource (for example semaphore) cannot be acquired without blocking and the non-blocking operation was requested. This error can also appear when trying to fetch the new incoming message and there is no message or when trying to get the response for the message sent and there is no response, etc.

YAMINOMEMORY

This is returned when memory allocation was not successful.

YAMICANNOTRESOLVEADDRESS

This is returned when the network address cannot be resolved to the 4-byte IP address, for example during domain registration.

YAMIBADADDRTYPE

This is returned when the network address resolution returned address of type other than `AF_INET`.

YAMICANNOTSTARTTHREAD

This is returned when there was a (platform-dependent) error while trying to start a new thread.

YAMITHREADERROR

This is returned when there was an unspecified threading error.

YAMISYNCHROERROR

This is returned when there was an unspecified (and platform-dependent) synchronization error.

YAMINETERROR

This is returned when there was an unspecified (and platform-dependent) network error.

YAMIOVERFLOW

This is returned for example while trying to push an element to the queue if it would break the queue's limits. The client code can see this error when trying to send a new message and there is no place in the sender thread's queue.

YAMIINTERNAL

This indicates a serious internal error, caused for example by synchronization or network problems in one of the *Agent's* threads. *Agent* returning this error cannot be used anymore and there is possible resource leak.

YAMINOHANDSHAKE

This indicates the lack of handshake in the network communication. The client code should never see this error.

6 YAMI Constants

The *YAMI* specification and the *YAMI* core library depend on some specific constant values, like the maximum length of the message name, the default size of connection pool, etc. Some of those values are defined by the *YAMI* specification and every implementation should comply to these values. The defaults are not so restricted – moreover, they are often arbitrarily chosen and other developers could have chosen them differently. The `yamilimits.h` header contains all the constants used by the *YAMI* core library.

The user is allowed to change the definitions in this header file and recompile the library, but it is important to note that *Agents* compiled with different limits may have communication problems.

6.1 Specification constants

```
#define MAXPARAMSETSIZE 1048576
```

Maximum size (in bytes) of the *Parameter Set* after packing to the sequence of bytes.

```
#define MAXPARAMSIZE      65536
```

Maximum size of a single parameter raw data. This value affects the variable-length types: strings, wide strings and binary objects.

```
#define MAXPARAMS        65536
```

Maximum number of parameters in a *Parameter Set*.

```
#define MAXNAMELEN       256
```

Maximum length of a name (message or object; it is also used to limit the length of domain name, although it is not defined in the specification).

6.2 Default values

```
#define YAMIDEFAULT_AGENTLEVEL      2
#define YAMIDEFAULT_OBJQMAXLENGTH  10
#define YAMIDEFAULT_OBJQMAXSIZE    1048576
#define YAMIDEFAULT_DISPATCHERS    1
#define YAMIDEFAULT_SENDERS        1
#define YAMIDEFAULT_SQMAXLENGTH    256
#define YAMIDEFAULT_SQMAXSIZE      1048576
#define YAMIDEFAULT_CONNPPOOLSIZE  10
#define YAMIDEFAULT_SENDTRIES      5
#define YAMIDEFAULT_HASSOCKET      1
#define YAMIDEFAULT_REUSEADDR      0
#define YAMIDEFAULT_HASRECEIVER    1
#define YAMIDEFAULT_HASWAKER       1
#define YAMIDEFAULT_MTUSER         1
```

These constants affect the default values that are assigned to the *Agent's Policies*.

6.3 Implementation choices

```
#define YAMINETLISTENBACKLOG          10
```

The backlog parameter passed to the `listen` system function, for the listening socket.

```
#define YAMIDELAYBEFORERERECONNECT    200
```

The maximum delay in ms before the attempt is made to reconnect after the existing connection was broken. This delay allows to distribute in time the load peaks when too many clients try to connect to the server at the same time; it has no influence in normal operation. The actual delay is a random value between zero and this constant.

7 Compilation options and possible problems

The following compilation options are provided:

7.1 Compilation for non-threaded platforms

It is possible to compile the library without the need to link with `pthread`. To do this, define the `YAMI_NO_THREADS` identifier as a compilation flag when compiling the library. This, of course, influences the functionality of the library:

- It is not possible to have additional threads in the *Agent*: the policies `dispatchers`, `senders`, `hasreceiver`, `haswaker` and `mtuser` should be set to 0 when the *Agent* is constructed. As a result, the explicit event loop is the only possibility to get the incoming messages processed.
- It is not possible to call any function that has blocking semantics implied. For example, it is not possible to wait until there is a message for given object (only non-blocking polling or direct dispatch is possible) or until there is a reply for a message. Any attempt to use the blocking functions will result in error `YAMINOSUCHSERVICE` or `YAMISYNCHROERROR`.
- It is not possible to use synchronization primitives provided with the library (and there would be hardly any reason to use them anyway).

This option is provided for those who want to experiment with *YAMI* on platforms that do not implement threading in the form of `pthread` library (for example, old Unix or some embedded systems) and for those who want minimal builds.

This flag should not be used when compiling Python or Tcl modules.

The `YAMI_NO_THREADS` flag can be used on MS Windows, but there is no reason to do so.