

YAMI C++ Wrappers Reference

Copyright © 2001-2008 Maciej Sobczak

Contents

1	Introduction	2
2	Parameter Set	3
3	Agent	8
3.1	Policies	8
3.2	New messages	10
3.3	Incoming messages	13
3.4	General API	17
3.5	Additional functions	26
4	Synchronization Primitives	27
5	Exceptions	29
5.1	Run-time exceptions	29
6	YAMI Constants	31
7	Compilation options and possible problems	32
7.1	Compilation for non-threaded platforms	32
7.2	Weak std::wstring support	32
7.3	Weak std::auto_ptr support	33

1 Introduction

The *YAMI C++ Wrappers* is a set of classes and free functions that are supposed to help C++ programmers to write modern C++ code using *YAMI* library. The wrappers replicate the whole functionality of this part of the *YAMI Core Library* that is supposed to be used directly by the end-user (by the programmer who wants to write programs using *YAMI*). The other part of the *YAMI Core Library* is not implemented in C++, but the wrappers are prepared for easy interfacing with the C code.

All the *YAMI C++ Wrappers* are declared in the `yami++.h` header and are placed inside the `YAMI` namespace.

Every function and method in this library can throw an exception in case of error, unless the function is documented as *no-throw*. See the respective part in this document for the exceptions description.

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.

```
class ParamSet
{
public:
    enum eType {
        eNoType = 0,
        eString = 1,
        eWString = 2,
        eInt = 3,
        eDouble = 4,
        eByte = 5,
        eBinary = 6
    };

    // constructor/destructor
    explicit ParamSet(int parcount);
    explicit ParamSet(void *handle, bool owner = true);
    ~ParamSet(); // throw()

    // copy
    ParamSet(const ParamSet &ps);
    ParamSet & operator=(const ParamSet &ps);

    // copy of one element
    void copyParameter(int positiondst, const ParamSet &ps,
                      int positionsrc);

    // clear one parameter
    void clear(int position);

    // setters

    void setInt(int position, int value);
    void setByte(int position, char value);
    void setDouble(int position, double value);
    void setBinary(int position, const char *buf, int bufsize);
    void setBinaryShallow(int position, const char *buf,
```

```

        int bufsize);
void setString(int position, const char *str);
void setString(int position, const std::string &str);
void setStringShallow(int position, const char *str);
void setWString(int position, const wchar_t *wstr);
void setWString(int position, const std::wstring &wstr);
void setWStringShallow(int position, const wchar_t *wstr);

// getters

int getParamCount() const;
eType getType(int position) const;
bool isASCII(int position) const;

int         getInt(int position) const;
char        getByte(int position) const;
double      getDouble(int position) const;
int         getBinarySize(int position) const;
void        getBinaryValue(int position,
                           char *buf) const;
const char * getBinaryBuffer(int position) const;
int         getStringLength(int position) const;
void        getStringValue(int position,
                           char *buf) const;
const char * getStringBuffer(int position) const;
void        getString(int position,
                     std::string &str) const;
int         getWStringLength(int position) const;
void        getWStringValue(int position,
                           wchar_t *buf) const;
const wchar_t * getWStringBuffer(int position) const;
void        getWString(int position,
                     std::wstring &wstr) const;

// handle operations

void * getHandle() const; // throw()
void * resetHandle(void *handle,
                  bool owner = true); // throw()
void swap(ParamSet &ps); // throw()
};

enum eType;

```

Note: this class is not intended to be a base class.

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

```
ParamSet(int parcount);
```

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

```
ParamSet(void *handle, bool owner = true);
```

Wraps the *Parameter Set* object around the already existing handle of type HPARAMSET. If the `owner` flag is true, the object will be destroyed in the wrapper's destructor.

```
ParamSet(const ParamSet &ps);  
ParamSet & operator=(const ParamSet &ps);
```

Creates a copy of the *Parameter Set*.
 In case of assignment, the previous object will be destroyed only if the wrapper owns it.

```
void copyParameter(int positiondst, const ParamSet &ps,  
                  int positionsrc);
```

Copies a parameter from one *Parameter Set* to the other. The source *Parameter Set* object is the one given in the `ps` parameter and "this" is the destination.

```
void clear(int position);
```

Clears the given entry.

```
void setInt(int position, int value);  
void setByte(int position, char value);  
void setDouble(int position, double value);  
void setBinary(int position, const char *buf, int bufsize);  
void setBinaryShallow(int position, const char *buf,  
                     int bufsize);  
void setString(int position, const char *str);  
void setString(int position, const std::string &str);  
void setStringShallow(int position, const char *str);  
void setWString(int position, const wchar_t *wstr);  
void setWString(int position, const std::wstring &wstr);  
void setWStringShallow(int position, const wchar_t *wstr);
```

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

```
int getParamCount() const;
```

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

```
eType getType(int position) const;
```

Gets the type of the parameter at a given position.

```
bool isASCII(int position) const;
```

Checks if the parameter is a plain, narrow ASCII string.

```
int    getInt(int position) const;
char   getByte(int position) const;
double getDouble(int position) const;
```

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

```
int          getBinarySize(int position) const;
void         getBinaryValue(int position, char *buf) const;
const char * getBinaryBuffer(int position) const;
```

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 `getBinaryValue` method.

```
int          getStringLength(int position) const;
void         getStringValue(int position, char *buf) const;
const char * getStringBuffer(int position) const;
void         getString(int position, std::string &str) const;
```

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 `getStringValue` function.

```
int          getWStringLength(int position) const;
void         getWStringValue(int position, wchar_t *buf) const;
const wchar_t * getWStringBuffer(int position) const;
void         getWString(int position, std::wstring &wstr) const;
```

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 `getWStringValue` function.

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.

```
void * getHandle() const; // throw()
void * resetHandle(void *handle, bool owner = true); // throw()
void swap(ParamSet &ps); // throw()
```

`getHandle` method returns a raw handle to the wrapped object. This handle is compatible with the `HPARAMSET` type. This method never throws.

`resetHandle` method allows to plug another `HPARAMSET` object in the existing wrapper. Never throws.

`swap` exchanges the underlying objects. Never throws.

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*.

3.1 Policies

```
class Policies
{
public:
    Policies(); // throw()
    ~Policies(); // throw()

    // setters // all throw()

    void setAgentLevel    (int  value); // 2
    void setObjQMaxLength(int  value); // 10
    void setObjQMaxSize   (int  value); // 1048576
    void setDispatchers   (int  value); // 1
    void setSenders       (int  value); // 1
    void setSQMaxLength   (int  value); // 256
    void setSQMaxSize     (int  value); // 1048576
    void setConnPoolSize  (int  value); // 10
    void setSendTries     (int  value); // 5
    void setHasSocket     (bool value); // true
    void setReuseAddr     (bool value); // true
    void setHasReceiver   (bool value); // true
    void setHasWaker      (bool value); // true
    void setMTUser        (bool value); // true
    void setAllowDuplex   (bool value); // true
    void setMaxDplxConns  (int  value); // 100
    void setReceiverIdle  (int  value); // 500

private:
    // copyright
    Policies(const Policies &);
    Policies& operator=(const Policies &);
};
```

Set of *Policies* that can be passed during the *Agent*'s creation.

Note: this class is not intended to be a base class.

Note: this class is *non-copyable*.

```
Policies(); // throw()
```

Fills the *Policies* structure with the default values. Never throws.

Meaning of the *Policies* (the default values are shown in comments above; they depend on the respective defaults in the Core Library):

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

setObjQMaxLength 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).

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

setDispatchers 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.

setSenders 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.

setSQMaxLength The maximum length of queue for sender thread.

setSQMaxSize The maximum size of the same queue.

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

setSendTries 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.

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

- setReuseAddr** 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.
- setHasReceiver** The flag stating if the separate receiver thread is needed. If set to `true`, the `setHasSocket` flag should be `true`, too.
- setHasWaker** 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.
- setMTUser** The flag stating if the user code is multi-threaded. If the value is `true`, the internal synchronization is provided so that all the *Agent's* functions are thread-safe. In single-threaded code, setting this value to `false` causes the *Agent* to bypass the internal synchronization whenever possible.
- setAllowDuplex** 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*.
- setMaxDplxConns** 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.
- setReceiverIdle** 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.

3.2 New messages

```
class Message
{
public:
    enum eStatus {
        ePosted,
        ePending,
        eReplied,
        eRejected,
        eUnknownObj,
        eOverflow,
        eNetError,
        eTimedOut,
        eRejectByAgent
    };
};
```

```

explicit Message(void *handle, bool owner = true);
~Message(); // throw()

void setTimeout(int timeout);
eStatus getStatus() const;
void wait() const;
std::auto_ptr<ParamSet> getResponse() const;

// handle operations

void * getHandle() const; // throw()
void * resetHandle(void *handle,
                  bool owner = true); // throw()
void swap(Message &msg); // throw()

private:
    // copyright
    Message(const Message&);
    Message& operator=(const Message&);
};

```

Note: this class is not intended to be a base class.

Note: this class is *non-copyable*.

The instances of this class can be created by the *Agent* object when the message is sent. See the `Agent::send` methods.

```
Message(void *handle, bool owner = true);
```

Wraps the message object around the already existing handle of type `HMESSAGE`. If the `owner` flag is true, the object will be destroyed in the wrapper's destructor.

```
enum eStatus;
```

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.

```
void setTimeout(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.

```
eStatus getStatus() const;
```

Retrieves the message's status.

```
void wait() const;
```

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 `setTimeout`, because without it, the `wait` can block *forever* (which can happen when the remote object crashes in the middle of processing).

```
std::auto_ptr<ParamSet> getResponse() const;
```

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

```
void * getHandle() const; // throw()
```

```
void * resetHandle(void *handle, bool owner = true); // throw()
```

```
void swap(Message &ps); // throw()
```

`getHandle` method returns a raw handle to the wrapped object. This handle is compatible with the `HMESSAGE` type. This method never throws.

`resetHandle` method allows to plug another `HMESSAGE` object in the existing wrapper. Never throws.

`swap` exchanges the underlying objects. Never throws.

3.3 Incoming messages

```

class IncomingMsg
{
public:
    explicit IncomingMsg(void *handle,
                        bool owner = true); // throw()
    ~IncomingMsg(); // throw()

    std::string getMsgName() const;
    std::string getObjectname() const;
    std::auto_ptr<ParamSet> getParameters() const;
    int getLevel() const;
    std::string getSourceAddr() const;
    int getSourcePort() const;
    std::string getEphemericalAddr() const;
    int getEphemericalPort() const;

    void eat();
    void reject();
    void reply();
    void reply(const ParamSet &paramset);

    void forward(const char *domainname, const char *objectname,
                const char *messagename);
    void forward(const std::string &domainname,
                const std::string &objectname,
                const std::string &messagename);
    void forward(const char *domainname, const char *objectname,
                const char *messagename,
                const ParamSet &paramset);
    void forward(const std::string &domainname,
                const std::string &objectname,
                const std::string &messagename,
                const ParamSet &paramset);

    void forwardAddr(const char *addr, int port, int level,
                    const char *objectname,
                    const char *messagename);
    void forwardAddr(const std::string &addr,
                    int port, int level,
                    const std::string &objectname,
                    const std::string &messagename);
    void forwardAddr(const char *addr, int port, int level,
                    const char *objectname,
                    const char *messagename,

```

```

        const ParamSet &paramset);
void forwardAddr(const std::string &addr,
                int port, int level,
                const std::string &objectname,
                const std::string &messagename,
                const ParamSet &paramset);

// handle operations

void * getHandle() const; // throw()
void * resetHandle(void *handle,
                  bool owner = true); // throw()
void swap(IncomingMsg &inc); // throw()

private:
    // copyright
    IncomingMsg(const IncomingMsg&);
    IncomingMsg& operator=(const IncomingMsg&);
};

```

Note: this class is not intended to be a base class.

Note: this class is *non-copyable*.

The instances of this class can be created by the *Agent* object when the new incoming message is received. See the `Agent::getIncoming` methods.

```
IncomingMsg(void *handle, bool owner = true);
```

Wraps the incoming message object around the already existing handle of type HINCMMSG. If the `owner` flag is true, the object will be destroyed in the wrapper's destructor.

```
std::string getMsgName() const;
```

Gets the name of the message.

```
std::string getObjectname() const;
```

Gets the name of the object, to which the message was sent.

It can be useful for *dispatch function* registered for different objects or for objects registered as `AnyObject`.

```
std::auto_ptr<ParamSet> getParameters() const;
```

Retrieves the *Parameter Set* that was sent together with the message.

This function can return NULL pointer value if the message was sent without parameters.

```
int getLevel() const;
```

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.

```
std::string getSourceAddr() const;
```

Retrieves the address of originating *Agent*. 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.

```
int getSourcePort() const;
```

Retrieves the port of originating *Agent*, see above for details.

```
std::string getEphemericalAddr() const;
```

Retrieves the address of connecting socket. 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.

```
int getEphemericPort() const;
```

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

```
void eat();
```

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).

```
void reject();
```

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

```
void reply();
void reply(const ParamSet &paramset);
```

Sends the reply, together with the *Parameter Set* (or without parameters) to the remote object.

```

void forward(const char *domainname, const char *objectname,
             const char *messagename);
void forward(const std::string &domainname,
             const std::string &objectname,
             const std::string &messagename);
void forward(const char *domainname, const char *objectname,
             const char *messagename,
             const ParamSet &paramset);
void forward(const std::string &domainname,
             const std::string &objectname,
             const std::string &messagename,
             const ParamSet &paramset);

void forwardAddr(const char *addr, int port, int level,
                const char *objectname,
                const char *messagename);
void forwardAddr(const std::string &addr,
                int port, int level,
                const std::string &objectname,
                const std::string &messagename);
void forwardAddr(const char *addr, int port, int level,
                const char *objectname,
                const char *messagename,
                const ParamSet &paramset);
void forwardAddr(const std::string &addr,
                int port, int level,
                const std::string &objectname,
                const std::string &messagename,
                const ParamSet &paramset);

```

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

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

`paramset` is a *Parameter Set* object that will be sent together with the message. It is also possible to send the message without the *Parameter Set*.

The “Addr” version allows to send new message with the destination address given explicitly, bypassing the *Agent*’s internal 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*.

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

```

void * getHandle() const; // throw()
void * resetHandle(void *handle, bool owner = true); // throw()

```

```
void swap(Message &ps); // throw()
```

`getHandle` method returns a raw handle to the wrapped object. This handle is compatible with the `HINCMMSG` type. This method never throws.

`resetHandle` method allows to plug another `HINCMMSG` object in the existing wrapper. Never throws.

`swap` exchanges the underlying objects. Never throws.

3.4 General API

```
extern const char *AnyObject;
```

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.

```
extern const char *ThisDomain;
```

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.

```
class PassiveObject
{
public:
    virtual ~PassiveObject() {}

    virtual void call(IncomingMsg &msg) = 0;
};
```

The class that is supposed to be a *base class* for all the *passive servants*. The `call` method will be called by the *Agent* when the incoming message arrives.

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

```
class Agent
{
public:
    // helper enum for level description
    enum eAgentLevel { eLevel1 = 1, eLevel2 = 2 };
};
```

```

// helper enum for connection options
enum eConnectionOptions { eNone = 0, eFixedDuplex = 1 };

// helper enum for return value from processEvent
enum eProcessEventResult
{
    eNothingToWaitFor = 0, // there was no socket for waiting
    eNoEvent          = 1, // there was no event to process
    eEventProcessed   = 2  // the event was processed
};

// constructor/destructor
explicit Agent(int port = 0);
Agent(int port, const Policies &policies);
explicit Agent(void *handle, bool owner = true); // throw()
~Agent(); // throw()

// domain set operations

void domainRegister(const char *name, const char *addr,
                   int port, int level, int options = 0);
void domainRegister(const std::string &name,
                   const std::string &addr,
                   int port, int level, int options = 0);
void domainUnregister(const char *name);
void domainUnregister(const std::string &name);

// object operation

enum eObjectType {
    ePolling,
    ePassiveSingleThreaded,
    ePassiveMultiThreaded
};

void objectRegister(const char *name,
                   enum eObjectType type,
                   PassiveObject *object);
void objectRegister(const std::string &name,
                   enum eObjectType type,
                   PassiveObject *object);

void objectUnregister(const char *name);
void objectUnregister(const std::string &name);

// incoming message operations

```

```

std::auto_ptr<IncomingMsg> getIncoming
    (const char *objectname, bool wait);
std::auto_ptr<IncomingMsg> getIncoming
    (const std::string &objectname, bool wait);

int getIncomingCount(const char *objectname);
int getIncomingCount(const std::string &objectname);

// outgoing message operations

std::auto_ptr<Message> send(const char *domainname,
    const char *objectname,
    const char *messagename);
std::auto_ptr<Message> send(const std::string &domainname,
    const std::string &objectname,
    const std::string &messagename);
std::auto_ptr<Message> send(const char *domainname,
    const char *objectname,
    const char *messagename,
    const ParamSet &paramset);
std::auto_ptr<Message> send(const std::string &domainname,
    const std::string &objectname,
    const std::string &messagename,
    const ParamSet &paramset);

std::auto_ptr<Message> sendAddr(const char *addr,
    int port, int level,
    const char *objectname,
    const char *messagename,
    int options = 0);
std::auto_ptr<Message> sendAddr(const std::string &addr,
    int port, int level,
    const std::string &objectname,
    const std::string &messagename,
    int options = 0);
std::auto_ptr<Message> sendAddr(const char *addr,
    int port, int level,
    const char *objectname,
    const char *messagename,
    const ParamSet &paramset,
    int options = 0);
std::auto_ptr<Message> sendAddr(const std::string &addr,
    int port, int level,
    const std::string &objectname,
    const std::string &messagename,

```

```

        const ParamSet &paramset,
        int options = 0);

void sendOneWay(const char *domainname,
               const char *objectname,
               const char *messagename);
void sendOneWay(const std::string &domainname,
               const std::string &objectname,
               const std::string &messagename);
void sendOneWay(const char *domainname,
               const char *objectname,
               const char *messagename,
               const ParamSet &paramset);
void sendOneWay(const std::string &domainname,
               const std::string &objectname,
               const std::string &messagename,
               const ParamSet &paramset);

void sendOneWayAddr(const char *addr,
                   int port, int level,
                   const char *objectname,
                   const char *messagename, int options = 0);
void sendOneWayAddr(const std::string &addr,
                   int port, int level,
                   const std::string &objectname,
                   const std::string &messagename,
                   int options = 0);
void sendOneWayAddr(const char *addr,
                   int port, int level,
                   const char *objectname,
                   const char *messagename,
                   const ParamSet &paramset, int options = 0);
void sendOneWayAddr(const std::string &addr,
                   int port, int level,
                   const std::string &objectname,
                   const std::string &messagename,
                   const ParamSet &paramset, int options = 0);

// additional functions

std::string getLocalAddress() const;
int getLocalPort() const;
eProcessEventResult processEvent(int timeout);

// handle operations

```

```

void * getHandle() const; // throw()
void * resetHandle(void *handle,
                   bool owner = true); // throw()
void swap(Agent &a); // throw()

private:
    // copyright
    Agent(const Agent &);
    Agent& operator=(const Agent &);
};

```

Note: this class is not intended to be a base class.

Note: this class is *non-copyable*.

```
enum eAgentLevel;
```

The convenience definitions for use (more verbose code) everywhere when the level parameter is required.

```
Agent(int port = 0);
Agent(int port, const Policies &policies);
```

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, then the *Agent* starts with the port number assigned by the operating system.

If no *Policies* object is used, the default values are used.

```
Agent(void *handle, bool owner = true);
```

Wraps the message object around the already existing handle of Core Library type HYAMIAGENT. If the *owner* flag is true, the object will be destroyed in the wrapper's destructor.

```

void domainRegister (const char *name, const char *addr,
                    int port, int level, int options = 0);
void domainRegister (const std::string &name,
                    const std::string &addr,
                    int port, int level, int options = 0);
void domainUnregister(const char *name);
void domainUnregister(const std::string &name);

```

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

The *options* parameter should be one of the values (or their bitwise sum) defined in the *eConnectionOptions* 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.

```
enum eObjectType;
```

Types of objects registered in the *Agent*.

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

ePassiveSingleThreaded 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.

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

```
void objectRegister(const char *name,
                   enum eObjectType type,
                   PassiveObject *object);
void objectRegister(const std::string &name,
                   enum eObjectType type,
                   PassiveObject *object);
```

Register 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 `AnyObject`. Only messages that have the destination object name matching with this parameter will be queued.

If the type is `ePolling`, then the `object` parameter is ignored. For passive objects, it's the address of the object, whose class derives from `PassiveObject` and implements the `call` method (it will be called for each incoming message).

```
void objectUnregister(const char *name);
void objectUnregister(const std::string &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.

```
std::auto_ptr<IncomingMsg> getIncoming
    (const char *objectname, bool wait);
std::auto_ptr<IncomingMsg> getIncoming
    (const std::string &objectname, bool wait);
```

This function gets the new incoming message for an object designated by the *objectname*.

If the `wait` parameter is `false`, this function will return immediately with the NULL pointer, when there is no incoming message for the given object and will *wait* if the `wait` is `true`.

```
int getIncomingCount(const char *objectname);
int getIncomingCount(const std::string &objectname);
```

This function gets the current length of the incoming message queue associated with the given object.

```
std::auto_ptr<Message> send(const char *domainname,
                           const char *objectname,
                           const char *messagename);
std::auto_ptr<Message> send(const std::string &domainname,
                           const std::string &objectname,
                           const std::string &messagename);
std::auto_ptr<Message> send(const char *domainname,
                           const char *objectname,
                           const char *messagename,
                           const ParamSet &paramset);
std::auto_ptr<Message> send(const std::string &domainname,
                           const std::string &objectname,
                           const std::string &messagename,
                           const ParamSet &paramset);

std::auto_ptr<Message> sendAddr(const char *addr,
                               int port, int level,
                               const char *objectname,
                               const char *messagename,
                               int options = 0);
std::auto_ptr<Message> sendAddr(const std::string &addr,
                               int port, int level,
                               const std::string &objectname,
                               const std::string &messagename,
                               int options = 0);
std::auto_ptr<Message> sendAddr(const char *addr,
                               int port, int level,
                               const char *objectname,
                               const char *messagename,
                               const ParamSet &paramset,
                               int options = 0);
std::auto_ptr<Message> sendAddr(const std::string &addr,
                               int port, int level,
                               const std::string &objectname,
                               const std::string &messagename,
                               const ParamSet &paramset,
                               int options = 0);
```

Sends new message.

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

It can be `ThisDomain`.

`objectname` is a name of the remote object.

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

`paramset` is a *Parameter Set* object that will be sent together with the message. It is also possible to send the message without the *Parameter Set*.

The “Addr” version allows to send new message with the destination address given explicitly, bypassing the *Agent*’s internal 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` should be one of the values defined in the `eConnectionOptions` enum type.

```
void sendOneWay(const char *domainname,
               const char *objectname,
               const char *messagename);
void sendOneWay(const std::string &domainname,
               const std::string &objectname,
               const std::string &messagename);
void sendOneWay(const char *domainname,
               const char *objectname,
               const char *messagename,
               const ParamSet &paramset);
void sendOneWay(const std::string &domainname,
               const std::string &objectname,
               const std::string &messagename,
               const ParamSet &paramset);

void sendOneWayAddr(const char *addr,
                   int port, int level,
                   const char *objectname,
                   const char *messagename,
                   int options = 0);
void sendOneWayAddr(const std::string &addr,
                   int port, int level,
                   const std::string &objectname,
                   const std::string &messagename,
                   int options = 0);
void sendOneWayAddr(const char *addr,
                   int port, int level,
                   const char *objectname,
                   const char *messagename,
                   const ParamSet &paramset,
                   int options = 0);
void sendOneWayAddr(const std::string &addr,
```

```
int port, int level,
const std::string &objectname,
const std::string &messagename,
const ParamSet &paramset,
int options = 0);
```

Sends new *one-way* message. See also the `send` and `sendAddr` methods.

```
std::string getLocalAddress() const;
```

Returns the local address of the *Agent* object (it does not need to have a listening socket) in the standard form *xxx.xxx.xxx.xxx*.

It can be useful for objects that need to pass their location to remote objects.

```
int getLocalPort() const;
```

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

If the *Agent* was created without the listening socket, this function throws an exception.

It can be useful for objects that need to pass their location to remote objects.

```
eProcessEventResult processEvent(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 `eNoEvent` value if there is no incoming connection. 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 method is supposed to be used in tight loops.*

```
void * getHandle() const; // throw()
void * resetHandle(void *handle, bool owner = true); // throw()
void swap(Message &ps); // throw()
```

`getHandle` method returns a raw handle to the wrapped object. This handle is compatible with the `HYAMIAGENT` type. This method never throws.

`resetHandle` method allows to plug another `HYAMIAGENT` object in the existing wrapper. Never throws.

`swap` exchanges the underlying objects. Never throws.

3.5 Additional functions

```
void easySend(const char *address, int port, int level,
              const char *objectname, const char *messagename);
void easySend(const std::string &address, int port, int level,
              const std::string &objectname,
              const std::string &messagename);
void easySend(const char *address, int port, int level,
              const char *objectname, const char *messagename,
              const ParamSet &ps);
void easySend(const std::string &address, int port, int level,
              const std::string &objectname,
              const std::string &messagename,
              const ParamSet &ps);
```

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.

```
void netInitialize(void);
void netCleanup(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).

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.

```
class Mutex
{
public:
    Mutex();
    explicit Mutex(void *handle, bool owner = true); // throw()
    ~Mutex(); // throw()

    void lock();
    void unlock();

    // handle operations

    void * getHandle() const; // throw()
    void * resetHandle(void *handle,
                       bool owner = true); // throw()
    void swap(Mutex &m); // throw()

private:
    // copyright
    Mutex(const Mutex &);
    Mutex& operator=(const Mutex &);
};
```

Note: this class is not intended to be a base class.

Note: this class is *non-copyable*.

The handle operations are compatible with the `HMUTEX` type.

```
class MutexHolder
{
public:
    explicit MutexHolder(Mutex &m);
    ~MutexHolder();

private:
    // copyright
```

```

    MutexHolder(const MutexHolder &);
    MutexHolder& operator=(const MutexHolder &);
};

```

Note: this class is not intended to be a base class.

Note: this class is *non-copyable*.

It is a RAII wrapper over the `Mutex` object with automatic locking (in the constructor) and unblocking (in the destructor).

```

class Semaphore
{
public:
    explicit Semaphore(int initval);
    explicit Semaphore(void *handle, bool owner = true); // throw()
    ~Semaphore(); // throw()

    void acquire();
    bool tryAcquire();
    void release();

    // handle operations

    void * getHandle() const; // throw()
    void * resetHandle(void *handle,
                      bool owner = true); // throw()
    void swap(Semaphore &s); // throw()

private:
    // copyright
    Semaphore(const Semaphore&);
    Semaphore& operator=(const Semaphore &);
};

```

Note: this class is not intended to be a base class.

Note: this class is *non-copyable*.

The handle operations are compatible with the `HSEMAPHORE` type.

```

void acquire();
bool tryAcquire();
void release();

```

Standard *p* and *v* operations on the semaphore object.

If the semaphore is not available, the `tryAcquire` method returns `false`.

```

void sleep(int timeout);

```

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

5 Exceptions

With very small exceptions (pun intended), all the *YAMI C++ Wrapper* methods and functions can throw exceptions.

All the exceptions are divided into two groups:

```
class LogicException : public std::logic_error
{
public:
    explicit LogicException(int cc);
    int getErrorCode() const;
};
```

These are programming errors that result from wrong usage of the *YAMI* library.

```
class RunTimeException : public std::runtime_error
{
public:
    explicit RunTimeException(int cc);
    int getErrorCode() const;
};
```

These are errors that result from factors that are outside of the *YAMI* library, for example network or threading problems. All the other *YAMI* exception classes derive from this class. Note, that for memory shortages the `std::bad_alloc` is thrown.

In both classes, the `getErrorCode` method returns the code that is compatible with the error code from the *YAMI Core Library*.

The standard `what()` method returns the human-readable error description.

5.1 Run-time exceptions

```
class BadAddress;
```

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

```
class OSErrors;
```

This is returned when there was a (platform-dependent) error.

The errors covered by this exception are threading or network problems.

```
class Overflow;
```

This is thrown 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.

```
class InternalException;
```

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

6 YAMI Constants

The *YAMI* specification and the *YAMI Core Library* 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. All those constants and defaults are taken from the *YAMI Core Library*. You can change them and recompile the library. Please refer to the *YAMI Core Library Reference* for more information.

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`s. 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 (or `false`, when appropriately) 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 exception `LogicException` (with notification "No such service.") or `OSError`.
- 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`s library (for example, old Unix or some embedded systems) and for those who want minimal builds.

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

Due to the shortcomings of some C++ compilers, it was necessary to provide some switches that enable the programmer to switch off some parts of the library or to use available alternatives. These switches exist in the form of preprocessor macros that are used for conditional compilations.

Currently, the following problems were identified:

7.2 Weak `std::wstring` support

The `g++ 2.95` compiler (it probably applies also to the 2.96 version) does not correctly support the `std::wstring` class, which is used in some members of the `YAMI::ParamSet` class. To switch those members off, and compile the rest of the wrappers, define the `YAMI_NO_WSTRING` identifier either directly in the code or as a compilation option like in the following example:

```
$ g++ -DYAMI_NO_WSTRING -c yami++.cpp
```

After that, the wide string feature will be accessible only through the C-style `wchar_t *` type.

7.3 Weak `std::auto_ptr` support

There is at least one known C++ compiler that does not provide the correct `std::auto_ptr` implementation. In order to overcome this problem, the alternative implementation is provided, see the `autoptr.hpp` file for its copyright notes. To use it, define the `YAMI_NO_AUTO_PTR` identifier for compilation of the C++ wrappers.