Libus++ Reference Manual

Paulo Guedes*
OSF Research Institute
1 Cambridge Center
Cambridge, MA 02142
pjg@osf.org

November 13, 1992

1 Introduction

This document describes the use of libus++, a C++ library to support the construction of clients and servers in the Mach 3 multi-server project. Some familiarity with the multi-server and MachObjects is expected.

The Mach 3 multi-server was initially implemented in MachObjects. This library aims at replacing MachObjects entirely with C++.

2 Overview of the programming model

The services exported by the servers are defined by C++ abstract classes. Single and multiple inheritance may be used to compose various classes to specify the interface of a service. The code of the clients uses the interface classes to represent the services offered by the servers. A service in a server is requested by invoking a method of an interface class.

The actual objects instantiated at the client's program are, in fact, subtypes of the interface classes. These derived classes contain a particular implementation. In the simple cases, they simply forward the invocations to the server, acting as a stub. In more sophisticated cases, they may keep local state and perform local processing.

An implementation class always derives from an interface class. In general, it uses multiple inheritance to also inherit implementation from another class. In a hierarchy with several layers of interface classes, an implementation class that implements the interface of level `i` inherits from the interface class of level `i` and from the implementation class that implements the interface of level `i-1`.

A server may itself be a client of other servers. A remote invocation service is responsible for instantiating the correct objects at the client's address space and interact with the underlying RPC service.

The model recognizes that some applications need to modify their view of the type of the objects at run-time and provides mechanisms to safely change at run-time the type of object references to a derived type.

*Author's present address: INESC, R. Alves Redol 9, 1000 Lisboa, Portugal, pjg@inesc.inesc.pt

There are three important classes in the library. Class `usTop` is the root of the hierarchy, all classes that need the functionality described here must derive from this class. Class `usRemote` derives from `usTop` and is the base class for all classes that contain methods that may be invoked remotely. Finally, class `usItem` is the base class of the multi-server specific class hierarchy and is used as base class for almost all other classes in the library. These three classes together implement almost all the functionality described here.

2.1 Interface classes

The declaration of an interface class is represented in Figure 1[1].

```
/* us_name_ifc.h */

class usName: public virtual usItem {
      public:
        DECLARE_MEMBERS_ABSTRACT_CLASS(usName);
REMOTE  virtual mach_error_t ns_resolve(char*, ns_mode_t, ns_access_t,
                                        usItem** , ns_type_t*, char*,
                                        int*, ns_action_t*) =0;
};

EXPORT_METHOD(ns_resolve);
```

Figure 1: Declaration of interface class `usName`

This figure represents the declaration of class `usName` as a C++ abstract class. Class `usRemote` is the base class for all classes that may be invoked remotely. Macro `DECLARE_MEMBERS_ABSTRACT_CLASS` declares a number of methods and class members that must be present in every class. The rest is simply the declaration of the methods exported by service `usName` as pure virtual functions.

Methods that may be invoked remotely are defined by macro `EXPORT_METHOD`. This macro defines a variable to be used by the RPC package. Macro `REMOTE` hints a possible notation for specifying the exported methods (currently this macro does nothing). Macro `EXPORT_METHOD` needs to appear only once in any class hierarchy, although multiple uses are not an error.

The recomended style is to use macro `EXPORT_METHOD` once in the interface class and the keyword `REMOTE` in every implementation class that exports the method remotely. This provides a clear and easy description of what methods are exported by a class.

The implementation of class `usName` is represented in figure 2.

This file contains type information about the parameters and is necessary to construct messages. Macro `DEFINE_ABSTRACT_CLASS` defines the implementation of some methods that must be present in all abstract classes. Macro `DEFINE_METHOD_ARGS` defines the type of the parameters of a method and must be used for all methods that may be invoked remotely. A call to this macro parallels the definition of a function interface with MiG.

A service provided by a server is requested by invoking a method of an interface class. From the perspective of a programmer who wants to use class `usName` as a client, file `us_name_ifc.h` contains all the information needed. File `us_name.cc` contains the information necessary for the run-time to send and receive messages for this class.

[1]The classes shown here are only examples. They may resemble actual classes used in libus++ but they are not exactly equal.

```
/* us_name.cc */

#include <us_name_ifc.h>

#define BASE usItem
DEFINE_ABSTRACT_CLASS(usName);

DEFINE_METHOD_ARGS(ns_resolve,"rpc: IN string; IN int; IN int; OUT * object<usItem>; OUT * int; O
```

Figure 2: Implementation of interface class usName

2.2 Implementation classes

Implementation classes derive from the interface classes. There are two kinds of implementation classes, client-side implementations, or proxy classes, and server-side implementations.

2.2.1 Proxy classes

The declaration of a proxy for class usName is represented in figure 3.

```
/* us_name_proxy_ifc.h */

#include <us_item_proxy_ifc.h>
#include <us_name_ifc.h>

class usName_proxy: public usName, public usItem_proxy {
      public:
        DECLARE_PROXY_MEMBERS(usName_proxy);
        usName_proxy() {};
        /*
         * Methods exported remotely
         */
REMOTE  virtual mach_error_t ns_resolve(char*, ns_mode_t, ns_access_t,
                                        usItem** , ns_type_t*, char*,
                                        int*, ns_action_t*);
};
```

Figure 3: Declaration of proxy class usName_proxy

Implementation class usName_proxy inherits the interface of class usName. It also inherits the implementation of usItem_proxy, which is the proxy implementation of usName's base class, usItem.

This file redeclares all the methods of usName, as is required in C++ when a derived class reimplements a method from the parent class. Macro DECLARE_PROXY_MEMBERS declares the members that must exist in all proxy classes. This macro is a different flavour of DECLARE_MEMBERS_ABSTRACT_CLASS.

The implementation of class usName_proxy is represented in figure 4. Macro DEFINE_PROXY_CLASS defines the implementation of several methods nedded by all proxy classes. Method init_class initializes the methods that can be invoked remotely. First it

calls the corresponding method on the base class and than it initializes all the remote methods in `usName_Proxy` with macro `SETUP_METHOD_WITH_ARGS`.

Methods acting as stubs (e.g. `ns_resolve()`) simply call function `outgoing_invoke()` to perform the RPC. The first parameter to this function is the identifier of the function, obtained with macro `mach_method_id`, the others are the parameters of the function.

```
/* us_name_proxy.cc */

#include <us_name_proxy_ifc.h>

DEFINE_PROXY_CLASS(usName_proxy);
DEFINE_CASTDOWN2(usName_proxy, usItem_proxy, usName)

void usName_proxy::init_class(usClass* class_obj)
{
        usName::init_class(class_obj);
        usItem_proxy::init_class(class_obj);

        BEGIN_SETUP_METHOD_WITH_ARGS(usName_proxy);
        SETUP_METHOD_WITH_ARGS(usName_proxy,ns_resolve);
        END_SETUP_METHOD_WITH_ARGS;
}

mach_error_t
usName_proxy::ns_resolve(char *path, ns_mode_t mode, ns_access_t access,
                         usItem** newobj,ns_type_t *newtype,char *newpath,
                         int *usedlen, ns_action_t *action)
{
        return outgoing_invoke(mach_method_id(ns_resolve),path,mode,access,
                               newobj,newtype,newpath,usedlen,action);
}
```

Figure 4: Implementation of proxy class `usName_proxy`

2.2.2 Server-side implementation classes

Server side implementation classes are constructed much in the same way as the implementation classes at the client side. Appendix B.3 represents one such class `dir`, a server side implementation of `usName` that also inherits implementation from class `vol_agency` internal to servers. This class uses basically the same constructs already seen in class `usName_proxy`, although different variants of some macros are used.

3 Declaration Files

A declaration file contains the C++ declaration of the class and normally has the extension `.h`. In libus++ these files are standard C++ files. Class `usTop` (file `include/top_ifc.h`) defines some macros that facilitate the declaration of classes. These macros declare a number of member fields and functions that must exist in all classes and declare the variables needed by the RPC subsystem.

Macro `DECLARE_MEMBERS` declares a number of members that must exist in all classes, in the form:

```
DECLARE_MEMBERS(class_name);
```

There are several variants of this macro:

DECLARE_MEMBERS declares the members of a "normal" class whose methods may be invoked remotely or whose references may be passed as parameters in remote invocations.

DECLARE_MEMBERS_ABSTRACT_CLASS same as `DECLARE_MEMBERS` for abstract classes.

DECLARE_LOCAL_MEMBERS same as `DECLARE_MEMBERS` for local classes whose methods are not accessible remotely.

DECLARE_PROXY_MEMBERS same as DECLARE_MEMBERS for proxy classes.

Macro `EXPORT_METHOD` defines an external variable that contains the method identifier for that method, and has the form:

```
EXPORT_METHOD(method_name);
```

4 Implementation Files

4.1 Generic definitions

Implementation files for interface classes contain the description of the parameters and are necessary to construct the messages between clients and servers.

Macro `DEFINE_CLASS` provides the implementation for the members declared with `DECLARE_MEMBERS` in the header file and has the form:

```
#define BASE base_class_name
DEFINE_CLASS(class_name);
```

There are several variants of this macro:

DEFINE_CLASS defines the members of a "normal" class whose methods may be invoked remotely or whose references may be passed as parameters in remote invocations. Assumes that `BASE` was defined with the name of the base class.

DEFINE_CLASS_MI same as `DEFINE_CLASS` for classes that inherit from multiple base classes. Method 'castdown' must be implemented "by hand" with DEFINE_CASTDOWN2 or DEFINE_CASTDOWN3.

DEFINE_ABSTRACT_CLASS equivalent to DEFINE_CLASS for abstract classes. These classes normally define an interface. This macro defines all the methods of these classes. Their implementation contains only the definition of the parameters of the methods (defined with macro DEFINE_METHOD_ARGS).

DEFINE_ABSTRACT_CLASS_MI same as DEFINE_ABSTRACT_CLASS for abstract classes with multiple base classes. Methods 'init_class' and 'castdown' must by defined "by hand". Use DEFINE_CASTDOWN2 or DEFINE_CASTDOWN3.

DEFINE_LOCAL_CLASS defines the methods for a class whose methods are not accessible remotely. These classes derive from 'usTop' and have garbage collection, synchronization, etc.

DEFINE_LOCAL_CLASS_MI same as DEFINE_LOCAL_CLASS for local classes with multiple base classes. Method 'castdown' must be defined "by hand" with DEFINE_CASTDOWN2 or DEFINE_CASTDOWN3.

DEFINE_PROXY_CLASS defines the methods for a proxy class.

4.2 RPC Interface

All classes that invoke or are invoked remotely must initialize the RPC dispatcher. This is achieved with method `init_class` as follows:

```
void class_name::init_class(usClass* class_obj)
{
        base_class1::init_class(class_obj);
        base_class2::init_class(class_obj);
        ...
        base_classN::init_class(class_obj);

        BEGIN_SETUP_METHOD_WITH_ARGS(class_name);
        SETUP_METHOD_WITH_ARGS(class_name,method1_name);
        SETUP_METHOD_WITH_ARGS(class_name,method2_name);
        ...
        SETUP_METHOD_WITH_ARGS(class_name,methodN_name);
        END_SETUP_METHOD_WITH_ARGS;
}
```

First all base classes have to be initialized by calling the same method, than macro `SETUP_METHOD_WITH_ARGS` is called for all methods of the class that may be accessed remotely. The calls to this macro are enclosed by calls to macros `BEGIN_SETUP_METHOD_WITH_ARGS` and `END_SETUP_METHOD_WITH_ARGS`.

Abstract classes must have an implementation file where the types of the parameters are defined. This is achieved with macro `DEFINE_METHOD_ARGS`, which has the form:

```
DEFINE_METHOD_ARGS(method_name, "parameter types");
```

All the information needed for packing and unpacking of the arguments for one method is specified using a simple ASCII string.

This string contains a method specification for global properties of the method itself, followed by a list of argument specifications, one for each argument in the order in which they appear in the method declaration. Either of these elements may be missing: if there is no method specification, the method defaults to a simple message with no response; if there are no argument specifications, the method will be invoked with no arguments. There is no argument specification for the object that is the target of the method invocation, since that argument is always present.

Each specification consists of a series of keywords or symbols separated by spaces. Extra spaces are ignored, unless they appear inside a keyword. Only the first letter of each symbol is significant,

and the rest of the word can be omitted. The case is significant for the first letter of each symbol. The parser for method argument strings is very primitive. Many syntax errors are not detected, and lead to unpredicatable results when the method is used at run-time. In addition, many combination of keywords are meaningless, and will also not be detected by the parser.

The method specification consists of one or more of the following keywords, followed by a colon (`':'`):

knownID indicates that the *method id* is well-known, and should be used to designate this method everywhere. Otherwise, the method name is used to designate it across address space boundaries.

rpc indicates that the method should be treated as an RPC, returning a 32-bit error code of type `mach_error_t` in addition to any OUT arguments that it may have. Otherwise, the method is forwarded as a single message with no reply and no success indication.

Each argument specification contains all the information needed to encode the data corresponding to one argument into a message. Unless otherwise specified, the system decides automatically whether or not to use the Mach IPC out-of-line transmission mode when applicable, based on the amount of data to transfer. The specification consists of the following elements, and is terminated by a semi-colon (`';'`):

- the **direction** indicates whether the argument is meaningful on entry to the method implementation, on exit, or both. It is a combination of the keywords `IN` and `OUT`. Arguments with the `IN` attributes are passed in the request message; arguments with the `OUT` are passed in the response message. If no direction is specified, `IN` is assumed.

- the **indirection count** indicates if the section of the caller's stack corresponding to the argument contains the data to be transmitted itself, or a pointer to that data, or a pointer to a pointer to that data. It is represented as zero, one or two `'*'` symbols.

  All three possibilities are valid for the `IN` direction, but for the `OUT` direction, only `'*'` and `'**'` are meaningful. In the first case, the caller must pass in a pointer to a pre-allocated memory area in which the desired data will be copied before the method invocation exits. In the second case, the caller must pass in a pointer to an empty pointer variable, and that variable will be initialized to point to a newly-allocated memory area containing the desired data. The user must explicitly destroy that memory area with `vm_deallocate()` when it is no longer needed.

  It is meaningless to specify the `OUT` keyword without an indirection count of at least one.

  For Mach IPC, the data is always sent out-of-line when double indirection is specified (`'**'`).

- the **copy flag** is meaningful only for the `IN` direction, with an indirection count greater than zero. It indicates that the pointer to the data, passed to the server-side object as a method argument, should point to an area of memory that is not automatically deallocated after the current method invocation exits. This flag is specified by the presence or absence of the `Copy` keyword.

  It is up to the user to destroy the memory with `vm_deallocate()` when it is no longer needed.

  For Mach IPC, this flag forces the data to be sent out-of-line.

- the **deallocation flag** indicates that the data to be transmitted must be deallocated from the sender's address space. This flag is specified by the presence or absence of the `Dealloc` keyword, and is meaningful both for `IN` and `OUT` arguments.

  For Mach IPC, this flag forces the data to be sent out-of-line, with the *dealloc* bit set.

- the **data type** indicates which data-type conversion may have to be performed in transit, and implies the size of each data element. It is one of the keywords `char`, `float`, `int` (32-bit), `object`, `word` (16-bit), `unstructured` (32-bit), `byte`, `string`, `ps` (send rights for port), or `pr` (receive rights for port).

  For type `string`, the argument is a pointer to a zero-terminated array of characters, unless the indirection count is one, in which case the argument is a pointer to a pointer to an array of characters. It is illegal to use type `string` with explicit double indirection.

  For type `object`, the argument is a pointer to an object instance, subject to the normal meaning of the indirection count. Details on the transmission of objects are given in the next section.

- the **data count** indicates how many data elements of the specified type constitute the argument to be transmitted. It consists of a number, enclosed in square brackets (`[n]`).

  If there is no data count specification, the argument is assumed to represent a single instance of the specified data type; otherwise, it is treated as an array.

  If the data count specification is the special construct `[*]`, the current argument is interpreted as a variable-size array, and the actual count of elements is taken from the next argument, which must be a simple integer and have the same direction attributes as the current argument. Note that this next argument must be described normally in the method arguments string (i.e. its specification is not implied by the presence of a variable-size array). It is illegal to specify such a variable-size array with an indirection count of zero, since, the next argument could then not be located on the stack.

  If a variable-size array is used as an `OUT` argument with an indirection count of one, it is the responsibility of the user to allocated enough space in the buffer area to receive the largest area possible. In this case, a maximum size can be specified by using a specification of the form `[*:n]`, where `n` indicates the maximum number of elements that can be copied in the buffer area pointed to by the call argument. This maximum size is ignored in all other cases.

- the **record size** applies only for arrays. It indicates that each element of the array (fixed- or variable-size) contains a specific number of data items of the type specified, and acts as a simple multiplier for the array size. It consists of a number enclosed in square brackets (`[n]`), just like a fixed-size array specification, but coming just after such a specification.

  If there is no record size specification, a size of one is assumed. It is not possible to define variable-size records, or heterogeneous records, whose elements are not all of the same type.

4.3 Using Objects as Method Arguments

When an object is used as an argument for a method used remotely, the actual information transmitted in the message is an object port.

When an object is received in a message, the port that represents is checked against a table of all external ports present in the address space. If there is already an object with that external port,

a new reference is taken for that object, and used as the method argument. Otherwise, a new object is instantiated, and associated with the specified port. The abstract class of the new object to be instantiated is specified by the method arguments string and is the name that appears enclosed in angle brackets (<class_name>) following the object keyword. The actual class of the object to be instatiated (proxy class) is defined by the server and is transmitted in the message. It must be a subclass of the abstract class defined in the method arguments string and its value is defined by a method called remote_class_name as follows:

```
char* class_name::remote_class_name() const
{
        return "proxy_class_name";
}
```

5 Cast-down

In some situations a pointer to a base class has to be converted to a derived class (cast-down). This is illegal in C++ with virtual base classes, therefore libus++ provides a mechanism for converting pointers to objects at run-time that is type-safe. All classes contain a method called castdown(p) that is used as follows:

```
base_class *p;
derived_class *q = derived_class::castdown(p);
```

Pointer p will be converted to class derived_class if and only if class derived_class is a derived class of base_class *and* the object pointed to by p is of type derived_class of a derived type of it. Otherwise this method returns the nil pointer.

A Normal Form for Method Arguments Strings

The following set of rules provides a normal-form specification for the syntax of method arguments strings. In this specification, non-terminals are enclosed in { and }, and terminals are enclosed in quotes. The symbol | represents an alternative within a production. A line with no left-hand side is a continuation line.

The special non-terminals {class-name}, {number} and {} correspond respectively to an identifier representing a class name, a positive integer number, and the null (epsilon) symbol.

```
{method-args-string}    :: {method-attributes} {argument-list}
{method-attributes}     :: {id-spec} {method-type-spec} ':' | {}
{id-spec}               :: 'knownID' | {}
{method-type-spec}      :: 'rpc' | {}
{argument-list}         :: {argument-spec} {argument-list} | {}
{argument-spec}         :: {direction} {indirection-count}
                           {copy-flag} {dealloc-flag}
                           {data-type} {size-spec} ';'
{direction}             :: 'IN' | 'OUT' | 'IN' 'OUT' | {}
{indirection-count}     :: {} | '*' | '**'
{copy-flag}             :: 'Copy' | {}
{dealloc-flag}          :: 'Dealloc' | {}
```

```
{data-type}             :: 'char' | 'float' | 'int' | 'word'
                        | 'byte' | 'unstructured' | 'string'
                        | {port-type} | {object-type}
{port-type}             :: 'ps' | 'pr'
{object-type}           :: 'object' {class-spec}
{class-spec}            :: '<'{class-name}'>'
{size-spec}             :: {data-count} {record-size} | {}
{data-count}            :: '['{number}']' | '[*]'
                        | '[*:'{number}']'
{record-size}           :: '['{number}']' | {}
```

Note that because of the particular parser used, many other grammars are also accepted. However, the grammar given here is sufficient to express all the features of method arguments strings. Note also that although all the strings accepted by this grammar are accepted by the parser, they do not necessarily correspond to meaningful arguments strings, and may cause run-time errors.

B Examples

B.1 Abstract classes

B.1.1 Declaration File us_name_ifc.h

```
/* us_name_ifc.h */

class usName: public virtual usItem {
      public:
        DECLARE_MEMBERS_ABSTRACT_CLASS(usName);
REMOTE  virtual mach_error_t ns_resolve(char*, ns_mode_t, ns_access_t,
                                        usItem** , ns_type_t*, char*,
                                        int*, ns_action_t*) =0;
REMOTE  virtual mach_error_t ns_create(char*, ns_type_t, ns_prot_t, int,
                                       ns_access_t, usItem**) =0;
REMOTE  virtual mach_error_t ns_list_types(ns_type_t**, int*) =0;
};

EXPORT_METHOD(ns_resolve);
EXPORT_METHOD(ns_create);
EXPORT_METHOD(ns_list_types);
```

B.1.2 Definition File us_name.cc

```
/* us_name.cc */

#include <us_name_ifc.h>

#define BASE usItem
DEFINE_ABSTRACT_CLASS(usName);

DEFINE_METHOD_ARGS(ns_resolve,"rpc: IN string; IN int; IN int; OUT * object<usItem>; OUT * int; O
DEFINE_METHOD_ARGS(ns_create,"rpc: IN string; IN int; IN * int[*]; IN int; IN int; OUT * object<u
DEFINE_METHOD_ARGS(ns_list_types,"rpc: OUT COPY DEALLOC ** int[*]; OUT * int;");
```

B.2 Proxy Classes

B.2.1 Declaration File us_name_proxy_ifc.h

```
/* us_name_proxy_ifc.h */

#include <us_item_proxy_ifc.h>
#include <us_name_ifc.h>

class usName_proxy: public usName, public usItem_proxy {
      public:
        DECLARE_PROXY_MEMBERS(usName_proxy);
        usName_proxy() {};
        /*
         * Methods exported remotely
         */
REMOTE  virtual mach_error_t ns_resolve(char*, ns_mode_t, ns_access_t,
                                        usItem** , ns_type_t*, char*,
                                        int*, ns_action_t*);
REMOTE  virtual mach_error_t ns_create(char*, ns_type_t, ns_prot_t, int,
                                       ns_access_t, usItem**);
REMOTE  virtual mach_error_t ns_list_types(ns_type_t**, int*);
};
```

B.2.2 Definition File us_name_proxy.cc

```
/* us_name_proxy.cc */

#include <us_name_proxy_ifc.h>

DEFINE_PROXY_CLASS(usName_proxy);
DEFINE_CASTDOWN2(usName_proxy, usItem_proxy, usName)

void usName_proxy::init_class(usClass* class_obj)
{
        usName::init_class(class_obj);
        usItem_proxy::init_class(class_obj);

        BEGIN_SETUP_METHOD_WITH_ARGS(usName_proxy);
        SETUP_METHOD_WITH_ARGS(usName_proxy,ns_resolve);
        SETUP_METHOD_WITH_ARGS(usName_proxy,ns_create);
        SETUP_METHOD_WITH_ARGS(usName_proxy,ns_list_types);
        END_SETUP_METHOD_WITH_ARGS;
}

mach_error_t
usName_proxy::ns_resolve(char *path, ns_mode_t mode, ns_access_t access,
                         usItem** newobj,ns_type_t *newtype,char *newpath,
                         int *usedlen, ns_action_t *action)
{
        return outgoing_invoke(mach_method_id(ns_resolve),path,mode,access,
                               newobj,newtype,newpath,usedlen,action);
}

mach_error_t
usName_proxy::ns_create(char *name, ns_type_t type, ns_prot_t prot,
                        int protlen, ns_access_t access, usItem** newobj)
{
```

```
        return outgoing_invoke(mach_method_id(ns_create),name,type,prot,
                               protlen,access,newobj));
}

mach_error_t usName_proxy::ns_list_types(ns_type_t **types, int *count)
{
        return outgoing_invoke(mach_method_id(ns_list_types),types,count);
}
```

B.3 Server-Side Classes

B.3.1 Declaration File dir_ifc.h

```
/* dir_ifc.h */

class dir: public usName, public vol_agency {
        struct mutex                lock;
      public:
        DECLARE_MEMBERS(dir);
        dir();
REMOTE  virtual mach_error_t ns_get_attributes(ns_attr_t, int*);
REMOTE  virtual mach_error_t ns_resolve(ns_path_t, ns_mode_t, ns_access_t,
                                        usItem** , ns_type_t*, ns_path_t,
                                        int*, ns_action_t*);
REMOTE  virtual mach_error_t ns_list_types(ns_type_t**, int*);
};
```

B.3.2 Definition File dir.cc

```
/* dir.cc */
#include <dir_ifc.h>

DEFINE_CLASS_MI(dir);
DEFINE_CASTDOWN2(dir, usName, vol_agency);

void dir::init_class(usClass* class_obj)
{
        usName::init_class(class_obj);
        vol_agency::init_class(class_obj);

        BEGIN_SETUP_METHOD_WITH_ARGS(dir);
        SETUP_METHOD_WITH_ARGS(dir,ns_resolve);
        SETUP_METHOD_WITH_ARGS(dir,ns_create);
        SETUP_METHOD_WITH_ARGS(dir,ns_list_types);
        END_SETUP_METHOD_WITH_ARGS;
}

dir::dir()
{
        mutex_init(&this->lock);
}

char* dir::remote_class_name() const
{
        return ``usName_proxy'';
}

mach_error_t
```

```
dir::ns_resolve(ns_path_t path, ns_mode_t mode, ns_access_t access,
                usItem **newobj, ns_type_t *newtype, ns_path_t newpath,
                int *usedlen, ns_action_t *action)
{
...
}
mach_error_t
dir::ns_create(ns_name_t name, ns_type_t type, ns_prot_t prot, int protlen,
               ns_access_t access, usItem **newobj)
{
...
}
mach_error_t dir::ns_list_types(ns_type_t **types, int *count)
{
...
}
```