Un recueil de papiers sur le systeme d'exploitation reparti a objets SOS

Marc Shapiro, Vadim Abrossimov, P. Gautron, S. Habert, Mesaac Makpangou

INRIA

UNITÉ DE RECHERCHE
INRIA-ROCQUENCOURT

Institut National
de Recherche
en Informatique
et en Automatique

Domaine de Voluceau
Rocquencourt
BP 105
78153 Le Chesnay Cedex
France
Tél. (1) 39 63 55 11

Rapports Techniques

N° 84

UN RECUEIL DE PAPIERS SUR LE SYSTÈME D'EXPLOITATION RÉPARTI À OBJETS SOS

Marc SHAPIRO
Vadim ABROSSIMOV
Philippe GAUTRON
Sabine HABERT
Mesaac MAKPANGOU

Mai 1987

UN RECUEIL DE PAPIERS SUR LE SYSTEME D'EXPLOITATION REPARTI ORIENTE OBJETS SOS

Marc Shapiro
Vadim Abrossimov
Philippe Gautron
Sabine Habert
Mesaac Makpangou

Résumé

SOS ou "SOMIW Operating System" est un projet de système d'exploitation réparti orienté objets pour le projet ESPRIT n°367 SOMIW ("Secure Open Multimedia Integrated Workstation", un environnement bureautique multimédia intégré). Dans ce système, la communication se base sur la notion de mandataire, ou objet représentant localement un service réalisé par un objet distant ou réparti.

Le présent recueil contient les documents suivants :

- **"SOS Programmer's Manual".**
- **"SOS : un système d'exploitation réparti basé sur les objets",**
- **"SOS: a distributed Object-Oriented Operating System",**
- **"A dynamic link editor for C++",**
- **"On the use of the dynamic link editor",**
- **"Exception Handling in C++ Programs,**

Le premier document décrit l'interface et l'utilisation du Prototype V1 de SOS, réalisé au-dessus d'Unix.

A COLLECTION OF PAPERS ON THE DISTRIBUTED OBJECT-ORIENTED OPERATING SYSTEM SOS

Abstract

SOS or the "SOMIW Operating System" is a distributed object-oriented Operating System project within Esprit n°367 SOMIW ("Secure Open Multimedia Integrated Workstation", an integrated multimedia office work environment). In SOS, communication is based on the proxy concept. A proxy is an object which represents, locally, some service implemented by a remote or distributed object.

This collection includes the following titles :

- **"SOS Programmer's Manual".**
- **"SOS : un système d'exploitation réparti basé sur les objets",**
- **"SOS: a distributed Object-Oriented Operating System",**
- **"A dynamic link editor for C++",**
- **"On the use of the dynamic link editor",**
- **"Exception Handling in C++ Programs,**

The first document describes the interface and the usage of SOS Prototype V1, implemented above Unix.

SOS Programmer's Manual

SOS Prototype Version 1

Vadim Abrossimov
Sabine Habert
Marc Shapiro

INRIA Bât. 11, B.P. 105, 78153 Le Chesnay Cédex, France

tel.: +33 1 39-63-53-25

e-mail: sos@corto.inria.fr

March 1987

NAME

intro – introduction to SOS

DESCRIPTION

An SOS prototype is composed of a kernel program, **sos**, an acquaintance service context **acq**, and whatever contexts you chose to run (see *sos(1), acq(1)*). A context is simply a specially-prepared Unix process (see *sosCC(1)*). Contexts may be run either automatically by naming them in the **predefContexts** file of the current directory, or by starting them (once **sos** and **acq** have finished initializing) from Unix, e.g. from the shell. Within a context multiple tasks my be run, using the C + + task primitives. Communication between contexts is possible if the requestor or *client* imports a *proxy* of the requested *service* (see *proxy(1), dynamic(1), makexport(1)*). Proxies communicate with their principal with the **crossInvoke** kernel primitive (see *crossInvoke(2), setTrapRef(2)*).

SOS programs must be prepared under Unix. The following preparation utilities are available: **sosCC** the C + + compiler (modified to allow dynamic linking and object migration, and with automatic linking to the appropriate libraries), and *makexport* a shell script to facilitate the preparation of exportable proxies.

There is also a short description of the use of the dynamic classes, in page *dynamic(1)*. Page *sosman(1)* explains the use of the SOS manual reader.

LIMITATIONS

This programmer's manual is for the SOS Prototype V1, running on a Sun-3 (release 3.0 or later). It is a prototype version intended to test the basic SOS ideas and run SOS programs. With respect to a fully functional SOS, it has the following limitations:

(1) runs on top of Unix, not the bare machine

(2) no Object Storage Service

(3) only a token Name Service

(4) No shared memory.

(5) No remote communication.

(6) Proxy interfaces are not checked at importation time.

(7) Inadequate protection of group membership, and of access to objects in general.

(8) No inter-object dependencies.

Point (6) means that you must be very careful, when importing an object, that its declared interface is the same as the actual one; otherwise unpredictable behaviour will occur.

In version V2, to be released in May 1987, we will add a Storage Service, a better Name Service, remote communication, and shared memory between contexts (within a group).

A port to the Metaviseur is also underway.

NAME

CC - C++ translator.

SYNOPSIS

CC [*option...*] *file...*

DESCRIPTION

CC (capital CC) translates C++ source code to C source code. The command uses *cpp*(1) for preprocessing, *cfront* and *cfront.dl* for syntax and type checking, and *cc*(1) for code generation.

CC takes arguments ending in

.c to be C++ source programs; they are compiled, and each object program is left on the file whose name is that of the source with **.o** substituted for **.c**.

.s to be assembly source programs; they are assembled, producing **.o** files.

CC interprets the following options:

-DL Run dynamic link cfront *(cfront.dl)*

+Z Allow non dynamic instances of dynamic classes (warnings are displayed).

-C Prevent *cpp* and *cfront* from removing comments.

-E Run only *cpp* on the .c files and send the result to standard output.

-F Run only *cpp* and *cfront* on the .c files, and send the result to standard output.

-Fc Like the **-F** option, but the output is C source code suitable as a .c file for *cc(1)*.

-.*suffix* Instead of using standard output for the **-E**, **-F** or **-Fc** options, place the output from each .c file on a file with the corresponding *.suffix*.

+V Accept regular C function declarations; use the */usr/include* directory **#include** files. Support for this option is not guaranteed in future releases.

+L Generate source line number information using the format "#line %d" instead of "# %d".

+x *file* Read a file of sizes and alignments. Each line contains three fields: a type name, the size (in bytes), and the alignment (in bytes). This option is useful for cross compilations.

+S Spy on *cfront*; that is, print some information on *stderr*.

See *ld*(1) for loader options, *as*(1) for assembler options, *cc*(1) for code generation options, and *cpp*(1) for preprocessor options (in the Unix manual).

FILES

file.c	input file
file..c	cfront output
file.o	object file
a.out	linked output
/lib/cpp	C preprocessor
cfront	C front end
cfront.dl	dynamic link C front end
/bin/cc	C compiler
/lib/libc.a	standard C library; see Section (3) in the *UNIX System V Programmer Reference Manual*
/lib/libC.a	C++ library
~sos/v1/lib/libD.a	dynamic linker library
/usr/include/CC	standard directory for **#include** files
/usr/include	standard directory for **#include** files when the **+V** option is used

SEE ALSO

sosCC(1), makexport(1), exception(1), dynamic(1)
in Unix manual: cc(1), monitor(3), prof(1), ld(1)
Bjarne Stroustrup, *The C + + Programming Language,* Addison-Wesley 1986.
B. W. Kernighan and D. M. Ritchie, *The C Programming Language,* Prentice-Hall 1978.

DIAGNOSTICS

The diagnostics produced by *CC* itself are intended to be self-explanatory. Occasional messages may be produced by the assembler or loader. No messages should be produced by *cc*(1).

BUGS

Some "used before set" warnings are wrong.

There is a (temporary) hole in the C + + type system allowing C + + programs to use C libraries. When a name is overloaded the first function of that name (only) can be linked to a library compiled by *cc*. Thus, the declaration

```
overload read(int,char*,int), read(vector*);
```

will allow the system call *read*(2) to be used together with user defined functions of the same name. Use of this facility may lead to unexpected behavior. For example, had the other **read()** been declared first, or had the system **read()** not been declared, then the user's **read()** would have been called by library functions like *scanf*(3).

Dynamic link errors (and internal bugs) begin with the message "Error dynamic link".

On Sun Release 3.2, it is not possible to compile in one pass. First compile individual with the -c option, then link them together with the -o option.

NAME

acq – the acquaintance service

SYNOPSIS

If file **predefContexts** of the current directory contains at least the line:

acq 1

then running **sos** automatically starts the acquaintance service.

Acq can also be started from the Unix shell by running **sos** in background, waiting for it to initialize, and then running **acq**.

DESCRIPTION

This is the binary file for the main acquaintance server on a machine.

FILES

./predefContexts

SEE ALSO

intro(1), sos(1), acquaintanceService(3)

NAME

dynamic – introduction to dynamic classes in SOS

SYNOPSIS

dynamic class [(*srchName*[,*srchName*...])] *clName*{
  // class declaration
};
dynamic [(*srchName*[,*srchName*...])] *clName instance;*
dynamic [(*srchName*[,*srchName*...])] *clName *ptr;*

DESCRIPTION

The keyword **dynamic** in a class or instance declaration means that the code may be loaded dynamically. The *srchName* is optional; it is interpreted differently under Unix and under SOS:

- In Unix, *srchName* is directly the name of a file containing the code for the class.
- In SOS, *srchName* is the name of a principal to be queried for a proxy. The principal must have previously registered this name with the Name Service. The principal must create an object of class **code**, with an indication of the file containing the code for the class (see *code(2)*). The proxy is imported and becomes the new instance.

In both cases, the default file name is the class name with the suffix .e appended; this file must have been previously compiled by **makexport(1)**; it must be in a directory named in the Unix environment variable **DLPATH**.

The actual migration and/or loading of code is performed by the procedure *dynamic__link*, which is called at instantiation time:

- before any instruction in the block for an **auto** variable;
- before allocation for an object allocated by **new**;
- before the execution of **main()** for static variables.

EXAMPLE

```
// in the header file
dynamic struct FOO {
  FOO ();
  int m ();
};

// in the export file, compiled with makexport
FOO::FOO () {
  printf ("constructor reached\n");
}

FOO::m () {
  return printf ("method m reached\n");
}

// in the import file, compiled with CC -DL
main () {
  dynamic ("test.e") FOO foo;
  foo.m ();
}
```

FILES

file.c	source file
file.e	export file

SEE ALSO

sosCC(1), makexport(1)
Bjarne Stroustrup, *The C + + Programming Language,* Addison-Wesley 1986.
B. W. Kernighan and D. M. Ritchie, *The C Programming Language,* Prentice-Hall 1978.
On the use of the dynamic link editor, in ~sos/v1/doc/dynalink

BUGS

The present version does not check the consistency of imported code with the importer; carefully check that the importer and the exporter were compiled using the same header files.

In "**dynalink (srchName) clName *ptr;**" the searchname is evaluated at instantiation time; its value may have changed since the declaration.

Both dynamic and non-dynamic instances of the same class are allowed, but cannot be mixed in the same source file. (To compile non-dynamic instances, use the **+Z** option of the compiler.)

The exported code for a dynamic class cannot be split into more than one source file. One source file canot contain the definition for more than one dynamic class.

When an object is imported, in SOS, the procedure **sosObject::initialize** is always applied, *after* the constructor, destroying any action of the constructor. It is currently not possible to define an alternative **initialize.**

The code for a dynamic class is imported only once; hence it is useless to use different .e file names for subsequent instantiations.

If the dynamic linker can't find a procedure, the whole SOS halts with an error message when the procedure is called.

Be patient about bugs, this a experimental version. Be careful with inline constructors.

DIAGNOSTICS

Dynamic link errors (and internal bugs) begin with the message "Error dynamic link".

NAME

makexport – C + + compilation of a dynamic class definition

SYNOPSIS

makexport [[+z *classname*] [-e *export_file*] *options*] *file*

DESCRIPTION

makexport compiles a C + + dynamic class *definition.* The result of the compilation is a file in export format, suitable for subsequent dynamic loading. This should be placed in a directory named in your Unix environment variable DLPATH. **Makexport** uses **CC** with dynamic linking.

All *options* are passed verbatim to CC, except:

-e is interpreted by *makexport* as the name of the export file. The default is the name of the file argument with suffix .c replaced by .e.

+z is used to make explicit the name of the exported class (necessary when more than one class is declared in .c file).

-g is not supported by makexport.

FILES

file.c input file
file..c temporary cfront output
file.o object file
file.e export file

SEE ALSO

sosCC(1), dynamic(1)
Bjarne Stroustrup, *The C + + Programming Language,* Addison-Wesley 1986.
B. W. Kernighan and D. M. Ritchie, *The C Programming Language,* Prentice-Hall 1978.
On the use of the dynamic link editor, in ~sos/v1/doc/dynalink/dl.tex

DIAGNOSTICS

Dynamic link errors (and internal bugs) begin with the message "Error dynamic link".

BUGS

Only one dynamic class may be defined in the single source file; its definition may not be split across source files. If other dynamic classes, other than the one defined, are declared, then the compiler gets confused; it is necessary to use the +z option to say explicitly which is the class being defined.

Be careful with inline constructors.

NAME

sos – start SOS

SYNOPSIS

sos

DESCRIPTION

Sos starts an SOS prototype under the UNIX operating system. **Sos** initiates the SOS kernel and then starts the contexts described by the **predefContexts** file. More contexts can be started as UNIX processes, e.g. from the shell (see *intro(1)*)

When the signal UNIX SIGINT arrives, **sos** aborts all active contexts and exits. There is two ways to abort an SOS execution, either by executing the Unix **exit** instruction in any of the contexts, or by sending SIGINT (e.g. by typing the interrupt character at the keyboad).

FILES

./predefContexts

must exist. It has one line per context to run automatically at start-up. A line has the following structure:

contextName number,

where *contextName* is the name of binary file of context (compiled by **sosCC -o ...**), and *number,* if present, is used to generate a (unique) context OID when the context is started by *sos.* **predefContexts** file should contain the following line:

acq 1

(see *acq(1)*).

BUGS

It is possible to have only one SOS running on one machine at one time.

When any of the contexts finishes, the SOS kernel and all the other contexts are terminated at once by a SIGINT signal.

The number of contexts whichmay be started on one machine at one time is limited by the number of open Unix file descriptors.

SEE ALSO

intro(1), sosCC(1), acq(1)

NAME

sosCC - compile C + + programs for SOS.

SYNOPSIS

sosCC [*options*] *file...*

DESCRIPTION

Compiles C + + files with the necessary options and libraries, to create SOS executables.

OPTIONS

Options are passed verbatim to *CC*. With no option or the *-o contextName* option, *sosCC* makes a binary file for an SOS context.

SEE ALSO

sos(1), makexport(1), dynamic(1)

NAME

sosman – read SOS manual pages.

SYNOPSIS

man [*section*] *title*

DESCRIPTION

sosman is a simple shell script which uses the Unix command **man(1)** to read the SOS manual pages from directory **~sos/v1/man**.

The SOS manual is divided into 3 sections:

1 describes Unix commands to compile and/or run SOS, or SOS applications.

2 describes the kernel and basic classes.

3 describes the Acquaintance Service and Name Service.

``SEE ALSO"

In Unix manual: man(1), man(5)

NAME

code – intermediate object for code of migrated object

DECLARATION (in code.h)

```
class code {
 public:
   . . .
   code (char*, char* ...);
   . . .
};
```

SYNOPSIS

```
/* in exporter, principal.c */
#include <sos/sos.h>
char * className = "proxyClass";
char * fileName = "proxyClass.e";
. . .
code * my_code = new code (className, fileName, 0);
```

ARGUMENTS

The *className* argument is the name of the class, which code is to be migrated.

The *fileName* argument is the name of file which contains the code. This file has been created by the **makexport** command (see Imakexport(1)). **The file name is searched in the Unix directories named in the environnement variable DLPATH.**

DESCRIPTION

In order to export proxies, a principal must first create a *code* object for the proxy's code. Within the execution of **giveProxy**, the principal uses **setCodeRef** to advertize the code object reference.

The **code** object is automatically migrated into the client context if and when needed, using the normal migration mechanisms of the Acquaintance Service.

BUGS

As we are not yet able to store code objects in SOS, one must create an instance of **code** for a proxy at each execution of SOS.

SEE ALSO

makexport(1), giveProxy(2), setCodeRef(2)

NAME

OID - object identifier class

DECLARATION (in OID.h)

```
class OID {
  . . .
public:
  . . .
  OID ();      /* default constructor */
       void
  allocate (); /* allocates a value to a null OID */
       void
  groupAllocate (unsigned long);
       int
  operator = = (OID);
       int
  operator != (OID);
       void
  print ();
  . . .
};
```

SYNOPSIS

```
#include <sos/sos.h>
. . .
OID oid1, oid2;
unsigned long seed;
. . .
oid1.allocate ();
oid2.groupAllocate (seed);
```

ARGUMENTS

The *seed* argument is an unsigned long less than 1 000 000. It is used as a seed to generate the group; the same seed is guaranteed to always generate the same OID.

DESCRIPTION

An OID is an Object IDentifier. Its possession gives no right on the object it identifies. Its scope is all of a SOMIW universe.

When it is created, an OID has a null value. The *allocate* procedure assigns it a new, unique value in the SOMIW universe. The *groupAllocate* assigns a new group OID value, see *group(2).*

To give your OID or your group OID to another object, see *giveMyOID(2).*

SEE ALSO

giveMyOID(2), group(2)

BUGS

The existence of the seed is a bug: it is impossible to guarantee that two independently created groups will actually have different group OIDs.

NAME

segmentDesc::allocate – create a segment and assign it to the segment descriptor

segmentDesc::free – destroy the segment referred by the segment descriptor

DECLARATION (in segment.h)

```
void
segmentDesc::allocate (long size = 0);
  void
segmentDesc::free ();
```

SYNOPSIS

```
#include <sos/sos.h>
 . . .
segmentDesc seg ( . . . );
seg.allocate (1000);
seg.free ();
```

DESCRIPTION

segmentDesc::allocate creates a segment of size *size* or, if *size* is 0 or omitted, of size **maxSize** (see **segmentDesc(2)**). The size of the segment can not be greater than **maxSize.**

segementDesc::free deallocates the segment (i.e., frees its memory) referred by the segment descriptor. Only the descriptor which was used to *allocate* the segment may be used to *free* it.

EXCEPTIONS

sgNotAllowed

operation is not in capability list (see **segmentDesc(2)**); or attempt to deallocate a segment which was not allocated using this segment descriptor.

sgBadSize

the **size** argument is greater than **maxSize** (see *segmentDesc(2)*), or both **size** and **maxSize** are 0 or omitted.

sgOutOfMemory

not enough memory to allocate the segment

SEE ALSO

segmentDesc(2)

NAME

segmentDesc::assign – assign a (sub)segment of an existing (sub)segment to the segment descriptor

DECLARATION (in sos.h)

```
void
segmentDesc::assign (byte* address, long size = 0);
void
segmentDesc::assign (segmentDesc* baseDesc, long offset = 0, long size = 0);
void
```

DESCRIPTION

The first form assigns a subsegment with the start address equal to *address,* and size equal to the *size* argument; or, if *size* is 0 or omitted, to *maxSize* (see *segmentDesc(2)).*

The second form assigns a subsegment with a start address equal to the start address of the (sub)segment referred by *baseDesc,* plus the *offset* argument. The size of the subsegment is equal to the *size* argument, or, if *size* is 0 or omitted, to the size of the *baseDesc* (sub)segment minus *offset.*

The size of the subsegment can not be greater than *maxSize* (see *segmentDesc(2)).*

EXCEPTIONS

sgNotLocal

operation is invoked for a remote segment

sgBadSize

the size of assigned subsegment is 0 or greater than *maxSize* (see *segmentDesc(2));* or (in the second form) the bounds of the requested subsegment are not included within the bounds of the original segment.

sgNotASegment

baseDesc does not refer to a valid segment

SEE ALSO

segmentDesc(2)

NAME

segmentDesc::copyTo – copy a (sub)segment into an other.

DECLARATION (in sos.h)

```
void
segmentDesc::copyTo (segmentDesc* targetDesc);
```

DESCRIPTION

Copy the contents of the (sub)segment referred by the segment descriptor into the (sub)segment referred by the *targetDesc*. The target (sub)segment must be already allocated and sufficiently large.

EXCEPTIONS

sgNotASegment

targetDesc descriptor or the source segment does not refer to a valid (sub)segment

sgBadSize

the size of the source (sub)segment is greater than the size of the target (sub)segment.

sgNotAllowed

the source descriptor does not have the **sgCopyFrom** capability, or the target descriptor does not have the **sgCopyTo** capability.

SEE ALSO

segmentDesc(2)

NAME

crossInvoke – cross-context invocation

DECLARATION (in sos.h)

```
class invokeMessage {
  . . .
 public:
   long opCode;    /* opaque for system */
};

class returnMessage { . . . };

   returnMessage*
crossInvoke (invokeMessage*, segmentDesc*[] = 0);
```

SYNOPSIS

```
#include <sos/sos.h>
class myInvokeMessage: public invokeMessage { ... };
class myReturnMessage: public returnMessage { ... };
myInvokeMessage* im;

segmentDesc *segs [] ;
long myOpCode ;
 . . .
// Fills the opCode field to select the remote procedure called
im -> opCode = myOpCode ;
myReturnMessage* rm = (myReturnMessage*) crossInvoke (im, segs);
 . . .
```

ARGUMENTS

The *im* argument is a pointer to an invocation message.

The *segs* argument is a vector of pointers to segments descriptors (see *segment(2)*). The vector must be terminated by a 0 pointer.

DESCRIPTION

Force a a process to continue its execution in the context of the principal of the calling object, as indicated by the *trapReference* in the latter's acquaintance descriptor (see *sosObject(2), setTrapRef(2)*). A bitwise copy of the invocation message is transfered and the descriptors for remote access to segements *(segs)* are passed. Only called in proxy !

CrossInvoke is to be called from within a proxy only.

RETURN VALUE

rm is a pointer to bitwise copy of the return message.

EXCEPTIONS

any exception
: if the principal terminates with an exception, then *crossInvoke* resignals the same one.

noPrincipal
: the principal is not found

LIMITATIONS

The size of invoke and return messages is limited by *MAXMESSAGE*. If they are larger, the result is undefined.

The Return message *(rm)* is valid only until the next **crossInvoke** call in the same task.

BUGS

The address of the calling object is taken from the contents of the stack. This only works if **crossInvoke** is called directly from the object's code (not from a procedure called by the object); also, the calling code may not be *inline*. One cannot crossIvnvoke into the same context as the caller.

SEE ALSO

proxy(2), stub(2), segment(2), setTrapRef(2)

NAME

segmentDesc::getAddr - get start address of (sub)segment

segmentDesc::getSize - get size of (sub)segment

segmentDesc::getMaxSize - get max. size of any (sub)segment possibly referred

segmentDesc::getCapability - get capability list

DECLARATION (in sos.h)

```
byte*
segmentDesc::getAddr ();
long
segmentDesc::getSize ();
long
segmentDesc::getMaxSize ();
segmentCapability
segmentDesc::getCapability ();
```

EXCEPTIONS

sgNotLocal

the **segmentDesc::getAddr** operation was attempted for a remote segment.

SEE ALSO

segmentDesc(2)

NAME

giveMyOID - give one OID of current instance to another accointance.

DECLARATION (in sosObject.h)

void
sosObject::giveMyOID (const sosObject*, const int);

SYNOPSIS

#include <sos/sos.h>
sosObject* *toObject;*
sosObject* *fromObject;*
int *index;*
fromObject->**giveMyOID** *(toObject, index);*

ARGUMENTS

The *toObject* argument is the address of an acquaintance (see *sosObject(2)),* to which an OID of *fromObject* is to be given. The *index* argument is the index of the desired OID, in the OID table of *fromObject*'s acquaintance descriptor.

DESCRIPTION

Add to the OIDs table in the acquaintance descriptor for *toObject,* the fromObject's OID designated by *index.*

EXCEPTIONS

Raises *notAnObject* if the address *toObject* does not refer to a valid acquaintance, *badIndex* if the *index* argument is negative or greater than MAX__OIDS - 1.

SEE ALSO

group(2), sosObject(2), reference(2), acquaintanceService(3)

NAME

giveProxy – user procedure called when a proxy is requested.

DECLARATION (in sosObject.h)

```
/* class of description for a proxy to export */
struct proxyDesc {
  ref proxy;  /* reference of the given proxy */
  int copy;   /* flag to say if the proxy's data is to be copied
                 (true) or moved (false, the default) to
                 the client context */
};
   virtual void
sosObject::giveProxy (const ref& client, proxyDesc& result);
```

SYNOPSIS

```
#include <sos/sos.h>
class myProxyClass: public sosObject { /* some class derived from sosObject */
  . . .
     void
  giveProxy (const ref& client, proxyDesc& result) {
       . . .          /* fill in the proxyDesc structure */
  };
  . . .
};
```

ARGUMENTS

The *client* argument is the reference of the client context (see *reference(2)*).

The *result* argument is a proxyDesc structure which is to be filled in.

DESCRIPTION

The *giveProxy* procedure is a user-written procedure. It should construct or select a proxy object for the *client*. The reference to this proxy object is returned in *result*. When this procedure returns successfully, the proxy is automatically migrated to the client context. This procedure should be implemented by each class of objects wishing to export proxies. It is called as part of the proxy migration protocol of the Acquaintance Service (see *acquaintanceService(3)*).

The acquaintance descriptor of the given proxy must contain its code reference and its trap reference, to be set by calling the **setCodeRef** and **setTrapRef** operations. The code object associated to the proxy is an instance of class **code** (see *code(2)*). Conversely, the principal may make its Trap Reference (or that of some other acquaintance) point to the proxy during the **giveProxy** operation, by calling **setTrapRef.**

The Acquaintance Service will automatically update the trap reference, when migrating the proxy (see *setCodeRef(2)*,setTrapRef(2)).

DEFAULT

The default, **sosObject::giveProxy**, always signals **refused** (see *sosObject(2)*).

RETURN VALUE

The *result*.**proxy** return value is the reference of the proxy to migrate.

The *result*.**copy** return value is a flag which indicates if the data segment of the proxy must be copied (true) or moved (false, the default), into the client context.

EXCEPTIONS

Signal **refused** to indicate unwillingness to create the requested proxy. Any exception which occurs is resignaled to the proxy requestor.

LIMITATIONS

We are not able to generate automatically the **giveProxy** operation. A library of standard **giveProxy** operations is predefined, containing currently the operations: **giveSelf, giveCopy.** For other cases, one must write one's own migration code (see proxy(2)).

SEE ALSO

code(2), proxy(2), reference(2), setCodeRef(2), setTrapRef(2), sosObject(2), acquaintanceService(3)

NAME

group – communication group, distributed object

DESCRIPTION

A group is defined as the set of acquaintances, across contexts, which carry a same OID:

- either all carry a same group OID,
- or one of the acquaintances gives its concrete OID to be used as a group OID by all the others.

SEE ALSO

OID(2), giveMyOID(2), sosObject(2), acquaintanceService(3), addGroupOID(3)

NAME

proxy – how to export, how to use, how to compile a proxy

SYNOPSIS

This is an example:

```
/* header file my__principal.h */
class my__principal : public sosObject {
  . . .
      void
   my__principal::giveProxy(const ref& , proxyDesc& ){
        . . .
   }
  . . .
};

/* header file my__proxy.h */
dynamic class myProxyClass: public sosObject {
  . . .
};

/* How to export a proxy, file my__principal.c) */
#include <sos/sos.h>
#include "my__principal.h"
#include "my__proxy.h"
code * code__proxy;
my__principal::my__principal(){
   ref proxyCodeRef;
   OID o;
   ulong ul;
        . . .
   o.groupAllocate (ul);
   // set its group OID
   AS->addGroupOID (this, o);
   // create a code object for future exportations of proxies
   code__proxy = new code ("myProxyClass", "my__proxy.e", 0);
   // register its reference with the name service
   NS->add ("my__principal", this);
}

   void
my__principal::giveProxy(const ref& client, proxyDesc& result){
   ref proxyCodeRef;
   . . .
   // decide whether to export or not
   if (notOK) raise(refused);
   // get reference of proxy code object
   AS->getReference (code__proxy, &proxyCodeRef);
   // create a non-dynamic instance of proxy locally
   myProxyClass* exportProxy = new myProxyClass (...);
   // give my group OID to the proxy
   giveMyOID (exportProxy, 1);
   // set proxy's code object reference
   exportProxy -> setCodeRef (proxyCodeRef);
```

```
    // if the principal wants to cross-invoke the proxy:
    this -> setTrapRef (exportProxy);
    // if the proxy needs to cross-invoke its principal:
    exportProxy -> setTrapRef (this);
    // prepare the proxy to migrate
    exportProxy -> giveSelf (result);
}

/* How to use a proxy (file my__client.c) */
#include "my__proxy.h"
main(){
    const char* the__principal = "my__principal";
    . . .
    dynamic (the__principal) myProxyClass* myProxy;
    . . .
    // importation and initialization of the myProxy
    // raises exception "refused","not found" or "noPrincipal" when fails
    myProxy = new myProxyClass;
    . . .
}
```

How to compile the different parts of the program (sample Makefile):

```
all: my__proxy.e my__principal my__client

my__proxy.e: my__proxy.h my__proxy.c
        makexport my__proxy.c

my__principal: my__principal.o my__proxy.o my__main.o
        sosCC -o my__principal my__principal.o my__proxy.o my__main.o

my__main.o: my__main.c my__principal.h
        sosCC -c my__main.c
my__principal.o: my__principal.h my__principal.c
        sosCC +Z -c my__principal.c
my__proxy.o: my__proxy.h my__proxy.c
        sosCC +Z -c myproxy.c

my__client: my__client.c my__proxy.h
        sosCC my__client.c
```

DESCRIPTION

An instance declared **dynamic** is assumed to be a proxy and the declarer is assumed to be its client. Instanciating a dynamic object triggers an importation. The *searchName* argument of the **dynamic** declaration is the symbolic name of a principal, registered with the Name Service. The principal is asked for a proxy, which is then imported, becoming the new instance.

The actual scenario, when executing the sample program above, is the following:

- Before anything, the principal registers its name with the Name Service, and creates a **code** object to hold the code of class *myProxyClass*. The code is loaded from a Unix file; the name of this file is declared in the **code** constructor, and evaluated relative to **DLPATH.**

At the time of the instanciation of *myProxy:*

- The Acquaintance Service queries the Name Service for the principal, using the searchname *"my_principal".*
- The principal's **giveProxy** procedure is called. This terminates:

  * either with an exception (which is resignaled to the client). In this case, the following steps are skipped. See the possible exception cases below.

  * or by normal termination, declaring a proxy *exportProxy* and its associated code object.
- The data for *exportProxy* and (if necessary) its code object are imported into the client's context.
- *myProxy* is instantiated by calling its constructor (but see bug notice below).
- The intialization procedure: **((sosObject*)** *myProxy* **) -> initialize()** overwrites *myProxy* with the data imported from *exportProxy.*

EXCEPTIONS

An importation request can fail with one of the following exceptions:

refused
if the principal's **giveProxy** procedure refuses to give a proxy to the client.

notFound
if the proxy reference returned by giveProxy does not designate an acquaintance of the principal's current context. In addition, the string **"Bad proxy reference"** is printed on the terminal.

noPrincipal
when there is no reference registered under *searchName* with the Name Service, or if the reference is obsolete and its update has not succeeded. In addition, the string **"Bad principal reference"** or **"Principal not registered in Name Service"** is printed on the terminal.

other
any other exception signaled by the principal's **giveProxy** procedure is resignaled to the importer.

COMPILATION

The code for a proxy must be compiled by the *makexport* command.

As it creates non dynamic instances of proxies, the code for a principal must be compiled using *sosCC* with the *+Z* option (see *sosCC(1)).* The code for a principal must be linked with the proxy code as in the synopsis above.

The code for the importer (the client) must be compiled with *sosCC.*

EXAMPLES

Please refer to the examples in ~sos/v1/examples.

SEE ALSO

crossInvoke(2), stub(2), giveProxy(2), dynamic(1), makexport(1), sosCC(1), code(2), setCodeRef(2), setTrapRef(2), acquaintanceService(3), addGroupOID(3), nameService(3)

BUGS

"Static" instances of dynamic classes are not allowed. If you declare an instance of a dynamic class "static" it cannot be initialized properly and behaviour is undefined.

No matter what class is declared for *myProxy*, it is always **sosObject::initialize** which is called. Any prior initialization performed by the constructor is overwritten by the imported data.

NAME

ref - acquaintance's reference class

DECLARATION

```
class ref{
    public:
        OID
    getOID();
        OID
    getCID();
        OID
    getSID();
        void
    print();
        int
    operator = = (ref);
        int
    operator! = (ref);
};
```

DESCRIPTION

A reference is an indication about the location of an acquaintance. It is constitued by an object identifier (see OID(2)) and a hint of location. The hint can be obsolete.

The *getOID* procedure returns the OID of the reference. This OID can be either a concrete OID, or a group one (see getReference(3)).

The *getCID* procedure returns the context OID in the hint of the reference.

The *getSID* procedure returns the site OID in the hint of the reference.

SEE ALSO

OID(2), getReference(3)

NAME

segmentDesc - descriptor for (sub)segment manipulation

DECLARATION (in sos.h)

```
class segmentDesc {
...
public:
   segmentDesc( long maxSize = 0, segmentCapability cap = 0 );
      void
   assign( byte* address, long size );
      void
   assign( segmentDesc* baseDesc, long offset = 0, long size = 0 );
      void
   allocate( long size = 0 );
      void
   free();
      void
   copyTo( segmentDesc* targetSegmentDesc );
      void*
   getAddr();
      long
   getSize();
      long
   getMaxSize();
         segmentCapability
   getCapability();
};
```

DESCRIPTION

A segment descriptor is used to refer (sub)segments.

Segment descriptor operations *segmentDesc::allocate(2)* and *segmentDesc::free(2)* allow to allocate/deallocate segments

Segment descriptor operation *segmentDesc::copyTo(2)* allows to copy a (sub)segment into another one

Segment descriptors may be transferred to another context by *crossInvoke(2)*. The tranferred descriptor is used in the target context to refer to the (sub)segment in the source context (which we call "remote (sub)segment"). The operations allowed in the target context may be limited by using the associated capability bit mask.

CONSTRUCTOR(S)

The *maxSize* argument allows to limit size of the (sub)segment which will be referred by the descriptor. If *maxSize* is 0, the size of referred (sub)segment is not limited.

The *cap* argument is a capability bit mask with the following fields:

sgAllocate
: *segementDesc::allocate(2)* operation allowed

sgFree *segmentDesc::free(2)* operation allowed

sgCopyFrom
: *segmentDesc::copyTo(2)* operation allowed

sgCopyTo
: the segment descriptor can be the target descriptor of a *segmentDesc::copyTo(2)* operation

The following operations can always be invoked and are not protected by a capability:

segmentDesc::getSize(2)

segmentDesc::getMaxSize(2)

segmentDesc::getCapability(2)

The following operations can never be invoked for a remote segment:

segmentDesc::getAddr(2)

segmentDesc::assign(2)

SEE ALSO

allocate(2), assign(2), copyTo(2), crossInvoke(2), getAddr(2), sosObject(2)

NAME

setCodeRef – update code reference of an acquaintance descriptor

DECLARATION (in sosObject.h)

void
sosObject::setCodeRef (ref&);

SYNOPSIS

#include <sos/sos.h>
ref& *itsCodeReference;*
sosObject **myObject;*
. . .
myObject **-> setCodeRef** *(itsCodeReference);*

ARGUMENTS

The *itsCodeReference* argument is a **reference** (see *reference(2))* to a **code** object representing the code for *myObject* (see *code(2)).*

DESCRIPTION

Fills the code reference field in the descriptor of the acquaintance, *myObject,* with the argument *itsCodeReference,* (see *acquaintanceService(3)).*

EXCEPTIONS

Raises *notAnObject* if *myObject* is not a valid acquaintance.

SEE ALSO

sosObject(2), reference(2), acquaintanceService(3)

NAME

setTrapRef – set Trap Reference of an acquaintance

DECLARATION (in sosObject.h)

```
void
sosObject::setTrapRef (const sosObject*);
```

SYNOPSIS

```
#include <sos/sos.h>
sosObject* toObject;
sosObject* myObject;
...
myObject -> setTrapRef (toObject);
```

ARGUMENTS

The *toObject* argument is the address of an acquaintance (see *sosObject(2)).*

DESCRIPTION

Fills the **trapReference** field of *myObject* with the reference to *toObject;* i. e. causes future cross-invocations performed by *myObject* to be directed to *toObject* (see *crossInvoke(2)).* It is verified that the acquaintances designated by *toObject* and *myObject* are in the same group (see *group(2)).* If the acquaintance designated by *toObject* migrates to an other context, any Trap Reference pointing to it, in *the current context* will automatically be updated to the new location.

EXCEPTIONS

Raises *notAnObject* if *toObject* does not refer to a valid acquaintance.

Raises *notSameGroup* if the acquaintances designated by *myObject* and *toObject* do not belong to a common group.

BUGS

Cross-invocation into the same context as the caller does not work.

SEE ALSO

crossInvoke(2), sosObject(2), reference(2), group(2), acquaintanceService(3)

NAME

sosObject – generic class for SOS acquaintances.

DECLARATION (in sosObject.h)

```
class sosObject {
   . . .
      sosObject*
   initialize (void*, int); /* always called after an import! */
 protected:
        virtual void
   giveProxy (const ref&, proxyDesc&);
        void
   setCodeRef (ref&);
        void
   setTrapRef (const sosObject*);
        void
   giveMyOID (const sosObject*, const int);
 public:
        virtual void
   stub(invokeMessage*, returnMessage*);
   sosObject();
  ~sosObject();
   giveSelf(proxyDesc&);
   giveCopy(proxyDesc&);
   . . .
};
```

DESCRIPTION

In SOS, objects known to the system must derive from class **sosObject**. This is necessary to be able to export proxies, to communicate by cross-invocation, etc.

When it is instanciated, an object derived from **sosObject** is automatically installed as an acquaintance in the current context. Conversely, the **sosObject** destructor removes the object from the list of acquaintances.

When an **sosObject** is imported, the procedure **initialize** is automatically applied to it; this procedure destroys any data initialized by the constructors!

The **giveProxy** procedure is called by the Acquaintance Service when a proxy of an acquaintance is requested. The default, **sosObject::giveProxy**, raises exception **refused**, meaning that the delegation of proxy is refused. Classes capable of exporting proxies should redefine **giveProxy**.

The **sosObject::setCodeRef** procedure fills the code reference field in the descriptor of the current instance.

The **sosObject::setTrapRef** procedure fills the trap reference field in the descriptor of the current instance.

The **sosObject::giveMyOID** procedure gives one OID of the current instance to another acquaintance.

The **sosObject::stub** procedure is the default stub for derived classes which don't redefine the name **stub(2)**. It always raises exception **refused**, meaning that all cross-invocations are refused. See

stub(2)

The **sosObject::giveProxy** procedure is the default for classes which don't redefine the name **giveProxy**. It always raises exception **refused** meaning that no proxy is exported. See *giveProxy(2).*

The **sosObject::giveSelf** procedure prepares the current acquaintance to migrate by filling its argument, **the proxyDesc** structure (see *giveProxy(2)).* It should be called by **giveProxy.**

The **sosObject::copySelf** procedure authorize the migration of a copy of the current acquaintance by filling its argument, the **proxyDesc** structure (see *giveProxy(2)).* It should be called by the **giveProxy.**

LIMITATIONS

"Auto" instances of **sosObjects** cannot be migrated; it is therefore more suitable to allocate **sosObject**'s by **new().**

For dynamic classes, instances are replaced by the imported object (by procedure **sosObject::initialize())** *after* application of the constructor. Hence, any initialization must be performed by some method, to be called by the application.

BUGS

Whatever the class of an imported object, it is always the procedure **sosObject::initialize** which is called! This procedure destroys any initializations done by the constructor.

SEE ALSO

stub(2), giveProxy(2), giveMyOID(2), setCodeRef(2), setTrapRef(2), acquaintanceService(3)

NAME

stub – principal's stub for the reception of cross-context invocations.

DECLARATION (in sosObject.h)

```
class invokeMessage {
 ...
public:
   long opCode;    /* opaque for system */
};
   virtual void
sosObject::stub (invokeMessage*, returnMessage*, segmentDesc**);
```

SYNOPSIS

```
#include <sos/sos.h>
class myClass: public sosObject { ... };
class myInvokeMessage: public invokeMessage { ... };
class  myReturnMessage: public returnMessage { ... };
 ...
   void
myClass::stub
 (invokeMessage* im, returnMessage* rm, segmentDesc *segs[]) {
   myInvokeMessage *mim = (myInvokeMessage*) im;
   myReturnMessage *mrm = (myReturnMessage*) rm;

   switch ( mim -> opCode ) { /* select action to perform */
      . . .
   }
}
```

ARGUMENTS

The *im* argument is a pointer to a bitwise copy of the *crossInvoke(2)* invocation message.

The *rm* is a pointer to the memory for the return message.

The *segs* argument is a vector of pointers to the descriptors for remote *segments(2)* access passed by *crossInvoke(2)*. The vector is terminated by a 0 pointer.

DESCRIPTION

A principal's *stub* function is invoked by the kernel, when it is the target of a *crossInvoke(2)* system call. The execution of the *stub* occurs (as if) in a new, dedicated task. During the *stub*'s execution only, the invoke message (**im*), the return message (**rm*) are accessible and the descriptors (*segs*) can be used for remote access to *segment; see(2)*. The return messsage is filled by **stub** and a bitwise copy of it is returned as the result of the *crossInvoke(2)* call.

DEFAULT

The default stub, **sosObject::stub**, always signals exception **refused.**

RETURN VALUE

A copy of the return message is returned to the invoker.

EXCEPTIONS

If **stub** terminates with any exception, it will be transmitted back to the caller of **crossInvoke.**

LIMITATION

The size of invoke and return messages is limited by *MAXMESSAGE.* If they are larger, the result is undefined.

SEE ALSO

crossInvoke(2), proxy(2), segment(2), sosObject(2), task(2), acquaintanceDescriptor(2)

NAME

unixChannel – base class for the Unix I/O in SOS

DECLARATION

```
class unixChannel {
 . . .
protected:
   short physChannel;
      void
   readReq ();
      void
   writeReq ();
public:
   unixChannel (int fd);
};
```

DESCRIPTION

The **unixChannel::unixChannel()** constructor takes a UNIX file descriptor *fd* as its argument. The constructor sets the value of the **physChannel** field equal to *fd*, meaning that subsequent I/O will take place using that file descriptor.

The functions **unixChannel::readReq()** and **unixChannel::writeReq()**, return, respectively, if and when the **physChannel** file descriptor is ready for a read or a write operation. The called task is rescheduled, if necessary.

SYNOPSIS

An example class, *readWrite* performing Unix-like read/write operations:

```
#include <sos/sos.h>

class readWrite: public unixChannel {
 public:
   readWrite (int fd) : (fd)    {}
      int
   read (char* buff, int len )  { readReq();
                                  return ::read (physChannel, buff, len);
                                }
      int
   write( char* buff, int len ) { writeReq();
                                  return ::write (physChannel, buff, len);
                                }
};
```

Now we can use it as follow:

```
 . . .
char buff[SIZE];
int fd = open( "/dev/tty", . . . );
readWrite myChannel (fd);
 . . .
int count = myChannel.read( buff, SIZE );
 . . .
```

If the task is suspended waiting for tty input, another task may run.

NAME

acquaintanceService – acquaintance service class.

DECLARATION (in acquaintanceService.h)

```
class acquaintanceService: public sosObject{
 public:
        void
   getReference (const OID&, ref*);
        void
   getReference (const sosObject*, ref*);
        void
   getReference (const short, ref*);
        void
   find (const ref&, radius, ref*);
        short
   getDescriptor (const sosObject*);
        short
   getDescriptor (const OID&);
        sosObject*
   getAddress (const OID&);
        sosObject*
   getAddress (const short);
        sosObject*
   isAnAcquaintance (const short dn);
        sosObject*
   isAnAcquaintance (const OID&);
        sosObject*
   isAnAcquaintance (const sosObject*);
        void
   getOID (const short, OID*);
        void
   getOID (const sosObject*, OID*);
        void
   getAllOIDs (const short desnum, OID[]);
        void
   getAllOIDs (const sosObject*, OID[]);
        radius
   locate ( const ref&);
        void
   stub (invokeMessage*, returnMessage*, segmentDesc**);
        void
   addGroupOID (const sosObject*, OID&);
};
extern acquaintanceService* AS;
```

DESCRIPTION

The acquaintance service is the manager for objects which need to be known by the system (to be able to export proxies, or to communicate by cross-invocation, etc.).

Objects known by the system are all derived from **sosObject** (see *sosObject(2)*) and are installed as acquaintances of current context by the acquaintance service.

The acquaintance service maintains information on objects, such as their address, their groups,

their *trap reference* (see *crossInvoke(2)*), etc. It manages proxy migration and object search (see *find(3)*) in co-operation with the kernel.

The acquaintance service is a distributed object. On each site, a main acquaitance service context is installed at the boot-time of SOS (see *sos(1), acq(1)*). Each newly created context inherits of a proxy of the acquaintance service, identified by the *AS* pointer. This proxy is the local Acquaintance Service of the new context, it is dedicated to the acquaintances of its current context. Each time an operation needs the intervention of several contexts (for exemple, search or migration), the *AS* passes the request to its principal.

SEE ALSO

sos(1), acq(1), crossInvoke(2), sosObject(2), stub(2), addGroupOID(3), find(3), getAddress(3), getAllOIDs(3), getDescriptor(3), getOID(3), getReference(3), isAnAcquaintance(3), locate(3)

NAME

addGroupOID – add an OID to an acquaintance's descriptor

DECLARATION (in acquaintanceService.h)

```
void
acquaintanceService::addGroupOID (const sosObject*, OID&);
extern acquaintanceService* AS;
```

SYNOPSiS

```
#include <sos/sos.h>
sosObject* addr;
OID& oid;
AS -> addGroupOID (addr, oid);
```

ARGUMENTS

The *addr* argument is the address of an acquaintance (see *sosObject(2)).*

The *oid* argument is an OID (see *OID(2)).*

DESCRIPTION

Adds the second argument, *oid,* to the array of Group OIDs in the descriptor of the acquaintance designated by the fisrt argument, *addr* (see *acquaintanceService(3)).*

EXCEPTIONS

Raises **tableFull** when the array of OIDs of the acquaintance descriptor is full, **notAnObject** if the address does not refer to a valid acquaintance.

LIMITATIONS

The number of OIDs for an acquaintance is limited by the size of the OIDs table in the descriptor.

SEE ALSO

sosObject(2), OID(2), acquaintanceService(3)

NAME

find – inter-context reference search.

DECLARATION (in acquaintanceService.h)

```
void
AcquaintanceService::find (const ref&, radius, ref*);
extern AcquaintanceService* AS;
```

SYNOPSIS

```
#include <sos/sos.h>
ref& ref1;
radius rad;
ref* ref2;
AS -> find (ref1, rad, ref2);
```

ARGUMENTS

The *ref1* argument is some obsolete reference (see *reference(2)), possibly obsolete.*

The *rad* argument is the radius of the search.

The *ref2* argument is a pointer to a reference structure.

DESCRIPTION

Searches, within the contexts included in the radius, for an acquaintance matching the reference specified by the first argument. Stores the updated reference in the location pointed by the *ref2* argument.

A *radius* is one of sameContext, sameWorkstation, sameOffice.

The search starts in the calling context and continues (until a match is found) in all the contexts situated within *radius* of the calling context.

RETURN VALUE

The return value *ref2* is a pointer to some reference.

EXCEPTIONS

When the search fails raises exception **notFound.**

SEE ALSO

reference(2)

NAME

getAddress – return the address of an acquaintance.

DECLARATION (in acquaintanceService.h)

```
sosObject*
acquaintanceService::getAddress (const OID&);
sosObject*
acquaintanceService::getAddress (short);
extern acquaintanceService* AS;
```

SYNOPSIS

```
#include <sos/sos.h>
short descriptorIndex;
OID& oid;
sosObject* addr;
addr = AS -> getAddress ( descriptorIndex );
addr = AS -> getAddress ( oid );
```

ARGUMENTS

The *descriptorIndex* argument is an index in the acquaintance descriptors table (see *acquaintanceService(3)*).

The *oid* argument is an object identifier, either a concrete OID or a group one (see *OID(3)*).

DESCRIPTION

Returns the address of an acquaintance of this context, indicated by the argument.

RETURN VALUE

The return value *addr* is a pointer to the acquaintance identified by the argument (see *sosObject(2)*).

EXCEPTIONS

Raises **notFound** if the indicated OID does not designate an acquaintance of this context, **notAnObject** if the index does not refer to a valid object.

SEE ALSO

sosObject(2), OID(2), acquaintanceService(3), getReference(3)

NAME

getAllOIDs - return the list of OIDs of an acquaintance.

DECLARATION (in acquaintanceService.h)

```
void
acquaintanceService::getAllOIDs (const short, OID[]);
void
acquaintanceService::GetAllOIDs (const sosObject*, OID[]);
extern acquaintanceService* AS;
```

SYNOPSIS

```
#include <sos/sos.h>
short descriptorIndex ;
sosObject* addr ;
OID oidList [MAX_OIDS];
AS -> getAllOIDs ( descriptorIndex, oidList );
AS -> getAllOIDs ( addr, oidList );
```

ARGUMENTS

The *addr* argument is a pointer to an acquaintance (see *sosObject(2)).*

The *descriptorIndex* argument is an index in the acquaintance descriptors table (see *acquaintanceService(3)).* The *oidList* argument is an array of OIDs (see *OID(2)).*

DESCRIPTION

Searches, in the current context, the list of OIDs of an acquaintance identified by the first argument. Stores the list in the array pointed by the *oidList* argument. The first OID in the array is the *concrete OID* of the acquaintance, the following are its *group OIDs.*

EXCEPTIONS

If the descriptor is unvalid or if the pointer is not the address of a local acquaintance, *getAllOIDs* raises the exception *notAnObject*.

SEE ALSO

sosObject(2), OID(2), acquaintanceService(3), getOID(3)

NAME

getDescriptor – return descriptor index of an acquaintance.

DECLARATION (in acquaintanceService.h)

```
short
AcquaintanceService::getDescriptor (const OID&);
short
AcquaintanceService::getDescriptor (const sosObject*);
extern acquaintanceService* AS;
```

SYNOPSIS

```
#include <sos/sos.h>
OID& oid ;

sosObject* addr ;
short descriptorIndex;

descriptorIndex = AS -> getDescriptor ( oid );
descriptorIndex = AS -> getDescriptor ( addr );
```

ARGUMENTS

The *oid* argument is an object identifier, either a concrete OID or a group OID (see *OID(3)*).

The *addr* argument is a pointer to an acquaintance (see *sosObject(2)*).

DESCRIPTION

Return the descriptor index of the acquaintance in the current context, identified by the argument.

RETURN VALUE

The return value *descriptorIndex* is an index in the acquaintance descriptors table of the current context (see *acquaintanceService(3)*).

EXCEPTIONS

Raises **notFound** if the indicated OID does not designate an acquaintance of this context, **notAnObject** if the address does not refer to a valid object.

SEE ALSO

sosObject(2), OID(2), acquaintanceService(3)

NAME

getOID – return OID of an acquaintance.

DECLARATION (in acquaintanceService.h)

```
void
acquaintanceService::getOID (short, OID*);
void
acquaintanceService::getOID (const sosObject*, OID*);
extern acquaintanceService* AS;
```

SYNOPSIS

```
#include <sos/sos.h>
short descriptorIndex ;
sosObject* addr ;
OID* oid ;
AS -> ( descriptorIndex, oid );
AS -> ( addr, oid );
```

ARGUMENTS

The *addr* argument is a pointer to an acquaintance (see *sosObject(2)*).

The *descriptorIndex* argument is an index in the acquaintance descriptors table (see *acquaintanceService(3)*).

The *oid* argument is a pointer to some OID (see *OID(2)*).

DESCRIPTION

Searches, in the current context, the *concrete OID* of an acquaintance identified by the first argument, and stores it in the location pointed by the *oid* argument.

EXCEPTIONS

If the descriptor is invalid or if the pointer is not the address of a local acquaintance, **getOID** raises **notAnObject**.

SEE ALSO

sosObject(2), OID(2), acquaintanceService(3), getAllOIDs(3)

NAME

getReference – return reference of an acquaintance.

DECLARATION (in acquaintanceService.h)

```
void
acquaintanceService::getReference (const OID&, ref*);
void
acquaintanceService::getReference (short, ref*);
void
acquaintanceService::getReference (const sosObject*, ref*);
extern acquaintanceService* AS;
```

SYNOPSIS

```
#include <sos/sos.h>
short descriptorIndex ;
OID& oid ;
sosObject* addr ;
ref* refp ;
AS -> getReference ( descriptorIndex, refp );
AS -> getReference ( oid, refp );
AS -> getReference ( addr, refp );
```

ARGUMENTS

The *descriptorIndex* argument is an index in the acquaintance descriptors table (see *acquaintanceService(3)).*

The *oid* argument is an object identifier. It can be either the concrete OID or a group OID of an acquaintance.

The *addr* argument is a pointer to an acquaintance (see *sosObject(2)).*

The *refp* argument is a pointer to a reference structure.

DESCRIPTION

Creates the acquaintance reference of the object designated by the first argument and stores it in the location pointed by *refp.* If the first argument is an OID, it will be the *identifier* field of the created reference (see *reference(2)).*

EXCEPTIONS

Raises **notFound** if the indicated OID does not designate an acquaintance of this context, **notAnObject** if either the index or the address does not refer to a valid object.

SEE ALSO

sosObject(2), reference(2), acquaintanceService(3)

NAME

isAnAcquaintance – check if an object is an acquaintance.

DECLARATION (in acquaintanceService.h)

```
sosObject*
isAnAcquaintance (const short);
sosObject*
isAnAcquaintance (const OID&);
sosObject*
isAnAcquaintance(const sosObject*);
extern acquaintanceService* AS;
```

SYNOPSIS

```
#include <sos/sos.h>
short descriptorIndex ;
OID& oid ;
sosObject* addr1 ;
sosObject* addr2 ;
addr2 = AS -> isAnAcquaintance ( descriptorIndex );
addr2 = AS -> isAnAcquaintance ( oid );
addr2 = AS -> isAnAcquaintance ( addr1 );
```

ARGUMENTS

The *descriptorIndex* argument is an index in the acquaintance descriptor table (see *acquaintanceService(3)).*

The *oid* argument is an object identifier, either a concrete OID or a group one see *OID(2)).*

The *addr1* argument is a pointer to an acquaintance (see *sosObject(2)).*

DESCRIPTION

Checks if the object designated by the argument is an acquaintance.

RETURN VALUE

Returns the address of the checked acquaintance if true, NULL if false.

EXCEPTIONS

None

SEE ALSO

sosObject(2), OID(2), acquaintanceService(3), getAddress(3)

NAME

locate - evaluate acquaintance location.

DECLARATION (in acquaintanceService.h)

```
    radius
acquaintanceService::locate (const ref&);
extern acquaintanceService* AS;
```

SYNOPSIS

```
#include <sos/sos.h>
ref& ref;
radius rad;
rad = AS -> locate (ref);
```

ARGUMENTS

The *ref* argument is a pointer to an object reference to be located.

DESCRIPTION

Returns the approximate distance between the caller and the location hinted at in the *ref* argument: one of **sameContext, sameWorkstation, sameOffice.**

RETURN VALUE

The *rad* argument is an indication of location.

EXCEPTIONS

None.

LIMITATIONS

The *ref* argument is not checked for validity.

NAME

nameService - name service class.

DECLARATION (in nameService.h)

```
class nameService{
  ...
 public:
        ref*
   lookup (char *);
        int
   add (char*, ref*, int = 0);
        int
   add (char*, sosObject*, int = 0);
   ...
};
extern nameService* NS;
```

SYNOPSIS

```
#include<sos/sos.h>
char* symbolic_name = "someName";
ref* reference;
sosObject* addr;
int result;
 result = NS -> add ( symbolic_name, addr );
AS -> getReference ( addr, reference );

 result = NS -> add (symbolic_name, reference );

 reference = NS -> lookup ( symbolic_name );
```

ARGUMENTS

The *symbolic_name* argument is a string of up to eighty characters (including the final `\0'), which names an acquaintance.

The *addr* argument is the address of an acquaintance in the current context (see *sosObject(2)).*

The *reference* argument is the reference of some acquaintance (see *reference(2)).*

The last argument says if the acquaintance is intended to be permanent or temporary.

DESCRIPTION

The name service is a minimal prototype, which allows to associate objects of the SOMIW universe with symbolic names.

It remembers the reference of acquaintances designated by a symbolic name during all the run-time of the SOS system.

Each newly created context inherits of an instance of the name service, identified by the *NS* pointer.

The *lookup* procedure returns the reference of the acquaintance designated by the *symbolic_name* argument, or the null value if it fails.

The *add* procedure registers the reference of the acquaintance designated either by the *addr* argument or the *reference* argument under the string pointed to by the *symbolic_name* argument. It returns zero in case of failure, one in case of success.

RETURN VALUES

The *reference* return value is a reference on some acquaintance in one SOMIW universe.

The *result* return value is a flag to say if the *add* procedure suceeded or not.

LIMITATIONS

As we are not yet able to store acquaintances in SOS, the scope of the name service is a SOS kernel execution. Each time the system is run, a new name service begins which creates a temporary Unix file, **/tmp/directory** to store symbolic names and their references. Each time the SOS system stops, this file is removed.

FILES

/tmp/directory stores SOS name/reference mappings.

BUGS

If the same symbolic name is used several times during one SOS execution, only the first registration will remain effective.

SEE ALSO

sos(1), sosObject(2), getReference(3)

SOS : un système d'exploitation réparti basé sur les objets*

Marc Shapiro
Vadim Abrossimov
Philippe Gautron
Sabine Habert
Mesaac Mounchili Makpangou

INRIA Bât. 11, B.P. 105, 78153 Le Chesnay Cédex, France
tel.: +33 1 39-63-53-25
uucp: ...!mcvax!inria!shapiro

16 mars 1987

Résumé

SOS, le système d'exploitation du projet Esprit SOMIW, est un système par objets, c'est-à-dire que tous les services et toute la communication sont vus comme des objets, qui s'accèdent par un interface procédural. L'accès d'un client à une ressource se fait par l'intermédiaire d'un objet propre au client et représentant la ressource, appelé mandataire. Un client ne voit d'une ressource, même répartie ou composée d'un groupe d'objets, que son seul mandataire. mandataire. Un éventuel protocole d'accès à la ressource est de même encapsulé derrière le mandataire ; il est programmé dans un objet de type protocole. Des mandataires différents peuvent être exportés vers des clients différents.

SOS est basé sur un petit noyau, complété par un gestionnaire d'objets et des objets système. L'invocation d'un objet est un appel de procédure simple. L'exécution du code associé à la procédure peut se poursuivre dans un autre espace mémoire, de façon transparente.

*A paraître dans "Techniques et Sciences Informatiques", printemps 1987.

1 Introduction

Le projet que nous présentons ici s'appelle SOS, pour "SOMIW Operating System". Il s'agit d'une des tâches du projet Esprit 367, "Secure Open Multimedia Integrated Workstation" ou SOMIW,[1] dans lequel sera construite une station de travail bureautique multi-média, intégrée sur un réseau, pour mettre en œuvre des documents comportant un mélange de texte, de graphiques, d'annotations vocales, d'images animées, etc. La tâche SOS conçoit, pour cet environnement, un système d'exploitation réparti original. Ce système est d'usage général, mais sa conception est fortement influencée par les besoins bureautiques. SOS supportera une émulation incomplète d'Unix mais il en diffère largement dans sa philosophie.

SOS est organisé autour du concept d'*objet* [Jones 79]. Un document multi-média par exemple peut être vu comme un ensemble structuré d'objets, de type paragraphe, graphique, annotation vocale, etc. Ces types dérivent tous de la même classe de base "unité textuelle"; ils peuvent donc tous être manipulés par les mêmes procédures (couper, ajuster, déplacer, etc.), bien que leurs actions pour chacune de ces procédures puisse être radicalement différente.

Cette approche doit favoriser l'intégration des différents développements dans le projet. Ainsi, un seul partenaire réalise les mécanismes de manipulation des "unités textuelles", qui sont réutilisés dans une base de données de documents, un éditeur, et un système de mise en page.

Le mécanisme des objets est aussi bien adapté aux systèmes répartis, puisque l'objet élémentaire est une unité de localisation, la communication se faisant seulement entre interfaces. De même, comme on verra plus loin que la notion d'objet est bien adaptée aux systèmes d'exploitation.

Au contraire de systèmes relativement monolithiques comme Unix, SOS est un système "ouvert" [Lampson 79], dans le sens qu'il est basé sur un petit noyau, la plupart des fonctions du système étant fournies à l'extérieur du noyau. Le noyau ne gère que les interruptions, les contextes (espaces de mémoire virtuelle), les processus, et la communication entre contextes. Il est complété par des objets système, comprenant les gestionnaires d'entrée-sortie, de la mémoire virtuelle, des disques et des fichiers, du nommage, de la communication répartie, etc.

Dans certains systèmes ouverts existants, comme par exemple le V-

[1] Les autres partenaires dans SOMIW sont : Bull-Transac (France, contractant principal), Sarin, Italtel Telematica et CSELT (Italie), Sobemap et CEN-SCK (Belgique), AEG Telefunken (République Fédérale Allemande), et INESC (Portugal).

System [Cheriton 84] ou Chorus-V1 [Zimmermann 84], la communication se fait par envoi de message dans un espace d'adressage plat. Nous pensons que cette approche, non structurée, atteint aujourd'hui ses limites. L'utilisation du mécanisme de *mandataire* (cf. § 3) permet de structurer la communication, en cachant à la fois le protocole et l'identité des partenaires éventuels.

2 Un système par objets

SOS est un *système par objets* : chaque entité visible (processus, boîte aux lettres, coupleur, catalogue, ensemble de versions d'un fichier, etc.) est un "objet", c'est-à-dire une boîte noire avec un interface bien défini. Un objet est *instance* d'une *classe*, laquelle définit son type, ses données internes, et son interface. En-dehors de l'instance, on n'a pas accès à ses données ; celles-ci ne peuvent être manipulées qu'en appelant les procédures de son interface. On peut créer autant d'instances d'une classe que nécessaire.

Une classe peut *dériver* d'une autre : ses instances auront des procédures compatibles (mêmes noms et types de paramètres), mais comporte peut-être une représentation interne différente ou des procédures supplémentaires. La classe dérivée est compatible avec sa "classe mère" du point de vue de la vérification des types.

Le système SOS n'en connaît pas l'organisation interne ni la signification des différentes procédures. De son point de vue, un objet est simplement un ou plusieurs segments de données attaché à une table de procédures, qui pointe elle-même vers du code compilé (cf. § 4.1 et la figure 2). Un objet est passif ; il est activé par l'*invocation* d'une des procédures de sa table. Ceci permet d'utiliser le langage fortement typé C++ [Stroustrup 85] pour la construction d'objets.

SOS est réalisé lui-même comme une collection d'objets système, qui s'exécutent au-dessus d'un noyau et d'un gestionnaire d'objets, dit "service d'accointances" (cf. § 4.4).

Une propriété importante des systèmes par objets est que les noms de procédure sont génériques. Ainsi, on peut accéder à des objets de construction différente par les mêmes appels. Par exemple, un fichier et une boîte aux lettres sont des objets diférents. Le code de lecture d'un enregistrement d'un fichier est différent de celui de la lecture d'un message ; mais si fichier et boîte aux lettres présentent tous deux un nom de procédure `lire` (par exemple), alors un programme pourra accéder à l'un ou à l'autre de façon transparente.

Contrairement à d'autres systèmes par objets comme Eden [Almes 85] ou Cronus [Schantz 86], mais de façon similaire à Apollo Domain [Rees 86], dans SOS le système connaît l'interface d'un objet ; l'utilisation d'un objet à la place d'un autre n'est permis qu'après que le système ait décrété que leurs interfaces sont compatibles.

Une des originalités de SOS est de permettre l'encapsulation d'une collection d'objets, participant à un même service, dans une nouvelle boîte noire dont les éléments sont indistingables de l'extérieur, même s'il sont répartis. Ceci est fait en obligeant toute communication entre un client et un service à transiter par un *mandataire*.

3 La notion de mandataire

Le "principe du mandataire" [Shapiro 86] stipule que, pour toute opération sur une ressource, un client doit invoquer un *mandataire*, c'est-à-dire un objet propre au client, et représentant, pour celui-ci, toute la ressource. Pour ce client, les seules opérations possibles sur la ressource sont les procédures du mandataire.

Les objets représentés par le mandataire sont appelés *mandants*. Ceux-ci sont dans des contextes ou sur des machines différentes de celle du client. L'ensemble des mandants et des mandataires d'une ressource donnée forment un seul objet réparti appelé *groupe*.

3.1 Exemple

Dans la figure 1 nous donnons l'exemple d'un service de courrier électronique réparti. Trois serveurs sur trois sites différents participent à ce service : deux boîtes aux lettres et un serveur de livraison.

Il n'y a pas d'interface usager spécifique, celui-ci étant fourni par un éditeur de texte classique. Pour que ceci soit possible, il suffit que l'interface au service "Courrier" soit le même que l'interface d'un fichier, c'est-à-dire que l'éditeur invoquera les mêmes procédures `ouvrir`, `lire` et `écrire` sur un objet "Fichier" que sur un objet "Courrier".

L'interface unique entre l'éditeur et les différents serveurs du service "Courrier" est constitué par le mandataire. Ainsi :

- le mandataire vérifiera la validité des commandes du client.
- Il regroupera ces commandes, et y répondra localement quand c'est possible en consultant ses données locales. Il peut y avoir par exemple

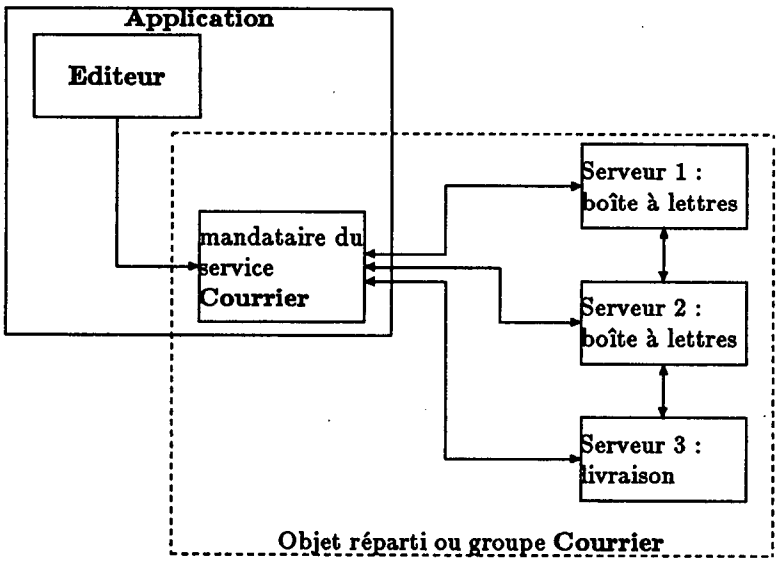


Figure 1 : Un mandataire pour un service de courrier réparti

un cache local des messages les plus récents.

- Il choisira un serveur approprié pour envoyer les commandes (choix entre boîte aux lettres active et livreur).
- Il communiquera avec les serveurs. Par exemple il transformera une commande de lecture d'un enregistrement, en provenance de l'éditeur, en un message à la boîte aux lettres active, demandant le prochain message.
- Il regroupera les résultats et les présentera au client.

Le mandataire cache la complexité du service réparti Courrier, le client n'ayant pas à connaître le choix du serveur ni le protocole de communication.

Entrons maintenant un peu plus dans le détail. Initialement, l'éditeur exécute une commande d'ouverture du genre `ouvrir("/mon/courrier","lecture")`. Le Service de Noms et le Service d'Accointances (c'est-à-dire le gestionnaire d'objets) recherchent un mandant répondant à ce nom ; supposons qu'en l'occurence il s'agit de "Serveur 1". Alors la procédure de *génération de mandataire* de "Serveur 1"

est invoquée, qui crée un objet (code et données). Puis cet objet est exporté vers le client, et installé dans son contexte de mémoire virtuelle (le rectangle en traits pleins de la figure 1), devenant ainsi le mandataire vers le service Courrier, propre à ce client. L'éditeur ne voit du service Courrier que son mandataire propre. Les opérations qu'il invoque sont des appels locaux des procédures visibles du mandataire. Au contraire, le mandataire a la visibilité des différents serveurs.

Notons que le mandant peut créer un mandataire différent selon les paramètres de l'ouverture (`"lecture"`), et des droits ou de l'identité du demandeur. Dans le cas présent, on ne permettra que la lecture des messages adressés à l'individu opérant l'éditeur (et pas ceux adressés à quelqu'un d'autre, ni la modification de ces messages).

3.2 Les propriétés d'un mandataire

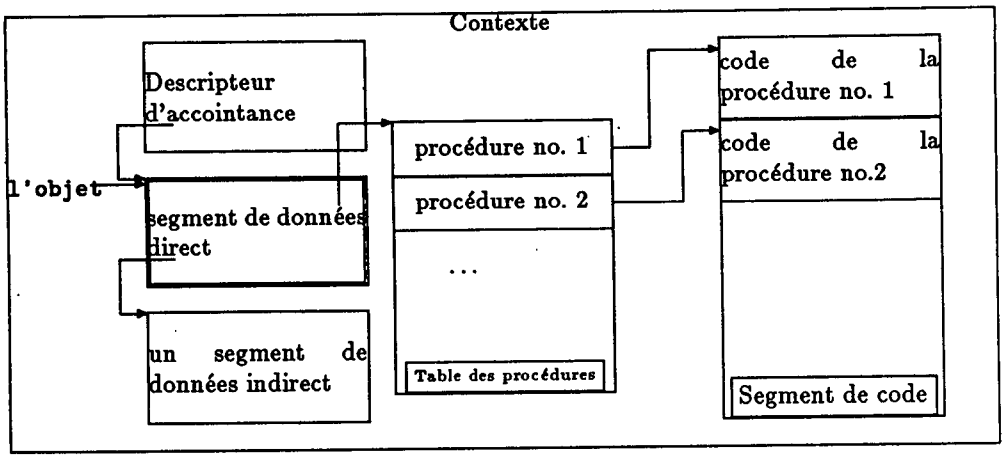
En conclusion, le mandataire est un outil pour l'encapsulation, car un client ne voit d'une ressource que son mandataire propre. A fortiori, le client ne peut pas connaître le protocole de communication entre mandataire et mandants.

Pour éviter que la décomposition d'un groupe ne puisse être dévoilée, SOS restreint les possibilites de communication. La communication entre contextes (par partage de mémoire, par messages, par un fichier commun) n'est permise qu'entres objets membres d'un même groupe.

Dans certains systèmes d'exploitation répartis, comme le V-System [Cheriton 84] or Chorus-V1 [Zimmermann 84], l'accès aux ressources se fait par une primitive d'envoi de message, selon un adressage indépendant de la localisation de la ressource. Cette approche a le mérite de la simplicité mais a l'inconvénient d'exposer la structure interne des services. Dans l'approche de SOS, ce n'est pas le client qui va au service (par messages), c'est le service qui va au client (en lui donnant un mandataire). Ainsi, toutes les ressources apparaissent locales et indécomposables. Bien entendu, le message reste un moyen de communication nécessaire, mais seulement entre mandants et mandataires.

4 Mécanismes de base

4.1 Le noyau



L'objet est connu des applications par l'adresse de son segment de données direct, noté ici `l'objet`. Il est connu du système par son entrée dans la Table des Descripteurs d'Accointance.

Figure 2 : Représentation d'un objet

Les objets gérés directement par le noyau sont les contextes, les processus, les segments, et les descripteurs d'objet. Ils sont décrits ci-après.

Les contextes sont des espaces de mémoire virtuelle indépendants. Un objet installé dans un contexte est appelé *accointance* de ce contexte.

Une accointance est connue du système par l'intermédiaire d'un descripteur d'accointance ; cf. figure 2. Celui-ci contient : un pointeur vers son premier segment de données, un pointeur vers sa table de procédures, et éventuellement la référence, dénommée `trapReference`, d'un objet du même groupe, dans un autre contexte (cf. § 4.2). Par défaut, la `trapReference` d'un mandataire désigne le mandant.

Plusieurs processus peuvent s'exécuter en parallèle dans le même contexte. Les objets étant passifs, un processus n'est pas lié à une accointance particulière.

4.2 Communication

La forme élémentaire de communication est l'invocation d'une procédure d'une accointance, indirectement à travers la table de procédures. Une entrée de cette table pointe souvent sur le code d'une procédure du contexte ; alors

il s'agit d'un appel procédural normal. Elle peut aussi contenir la primitive d'invocation inter-contexte. Dans ce cas, le processus appelant continuera son exécution dans un autre contexte, invoquant l'objet indiqué dans la `trapReference` de son descripteur.

Les mécanismes de mémoire virtuelle sont mis à profit pour optimiser le coût de la communication entre contextes (partage de mémoire, recopie à l'écriture, etc.).

Il n'y a pas de mécanisme spécifique de communication distante, mais celle-ci est possible via un objet de type *protocole*, ayant accès au réseau. En général un tel objet réalise une sorte de circuit virtuel avec ses pairs attachés aux objets distants du même groupe. Le protocole par défaut étend simplement l'invocation inter-contexte à l'appel de procédure distant sur une connexion fiable. D'autres protocoles, dictés par les applications, et prises dans une bibliothèque de protocoles, peuvent sans problème être substitués à ce dernier, puisque le protocole reste interne au groupe (cf. § 3.2).

4.3 Dépendances entre objets

Dans SOS, toutes les relations existant entre objets, à l'exécution, sont explicites. En effet, il ne peut exister une dépendance entre deux contextes que si une la `trapReference` de l'un pointe dans l'autre. De même, une relation entre deux machines est indiquée par un objet de type protocole les reliant.

Cette information permet à SOS de maintenir un graphe des dépendances entre objets. La destruction d'un objet est signalée automatiquement à tous ses dépendants. Ce signal sera correctement ordonné par rapport aux autres messages. La procédure par défaut de traitement du signal de dépendance détruit le dépendant à sont tour ; cette procédure peut être remplacée. Par défaut, un mandant et un mandataire dépendent mutuellement l'un de l'autre.

Un exemple illustrera l'utilité de la gestion des dépendances. Soit V un objet gestionnaire de verrous pour le compte d'un autre objet O. V dépend de O. La procédure de traitement du signal de destruction de O relâchera les verrous attachés à O.

4.4 Le service d'accointances

Le service d'accointances est le gérant des objets, en coopération avec le noyau. C'est lui qui maintient les informations sur les objets comme leur adresse, leur dépendance, leur groupe, leur `trapReference`.

Le service d'accointances est lui-même un objet réparti. Il y en a un représentant "principal" dans chaque machine, installé à l'initialisation de celle-ci, plus un représentant "secondaire" dans chaque contexte, installé à sa création. Ils sont tous mandataires les uns des autres.[2] Un service d'accointances secondaire s'occupe des accointances de son contexte. Lorsqu'une opération fait intervenir plusieurs contextes (par exemple, recherche ou migration) elle est relayée par les représentants principaux des machines concernées.

L'installation d'un mandataire comporte une phase de création par le mandant, puis de migration vers le contexte du client. C'est le service d'accointances qui supervise ces opérations. D'abord, il recherche le mandant (par un numéro unique) ; puis invoque, dans le contexte de celui-ci, sa procédure de génération de mandataire. Cette procédure retourne le segment de données initial et un descripteur. Ces derniers sont alors transportés dans le contexte demandeur. Ensuite, le code (dont la référence se trouve dans le descripteur) est importé si nécessaire ; l'édition de liens se fait dynamiquement. Finalement, la procédure d'initialisation de l'objet est invoquée.

Pour plus de détails, se referer à [Habert 86].

5 Etat courant du projet

Le système est écrit en C++ [Stroustrup 85]. Il fonctionnera sur machine nue ; sa mise en œuvre se base sur l'expérience acquise dans le projet Chorus [Zimmermann 84].

Les spécifications fonctionnelles du système sont écrites. Une simulation du noyau, utilisant Unix, est en cours de réalisation. Le service d'accointances fonctionne. Nous savons charger un objet et en faire l'édition de liens dynamique. Le prototype de SOS sur machine nue devrait fonctionner à la fin de 1987, et une réalisation complète est prévue pour la fin 1988.

Des modifications au compilateur C++ sont en cours, afin d'y intégrer l'édition de liens dynamique et en simplifier l'utilisation. D'autres modifications à la chaîne de production sont prévues, pour pouvoir vérifier, au moment de d'une importation, que l'interface du code importé est bien compatible avec la façon dont il est utilisé par le code déjà présent.

[2] Ceci est possible car le système sous-jacent ne fait pas de différence entre mandataire et mandant. Le seul mécanisme nécessaire pour faire fonctionner ces notions est celui de groupe.

Un des buts initiaux de notre projet est de permettre l'exécution simultanée d'activités de nature différente dans le même système, dont le spectre s'étend des activités temps-réel (transport de voix et d'images animées) aux transactions atomiques et aux activités à haute disponibilité (procédures bureautiques fiables). Les objets de type protocole assureront que les activités de type différent ne rentrent pas en conflit, tout en permettant à chaque activité de ne payer que le prix de ses propres mécanismes. La réalisation de cette facilité fera l'objet de nos recherches futures.

Bibliographie

[Almes 85] Guy Almes, Andrew Black, Edward Lazowska, Jerry Noe. The Eden system: a technical review. *IEEE Transactions on Software Engineering*, SE-11(1), janvier 1985.

[Cheriton 84] David R. Cheriton. The V-Kernel, a software base for distributed systems. *IEEE Software*, 1(2):19–42, avril 1984.

[Habert 86] Sabine Habert. *La migration dans les systèmes d'exploitation répartis.* Rapport de DEA de systèmes informatiques, Université Paris VI, juin 1986.

[Jones 79] A. K. Jones. The object model: a conceptual tool for structuring software. Dans : R. Bayer, R. M. Graham, & G. Seegmuller, éditeurs, *Operating Systems, an Advanced Course*, Springer-Verlag, New York (USA), 1979.

[Lampson 79] B. Lampson, R. F. Sproull. An open operating system for a single-user machine. Dans : *Proc. 7th Symp. on O.S. Principles*, pages 98–105, 1979.

[Rees 86] Jim Rees, Paul H. Levine, Nathaniel Mishkin, and Paul J. Leach. An extensible I/O system. In *Proc. Usenix Summer Conference*, Atlanta, Georgia (USA), 1986.

[Rashid 81] R. Rashid, G. Robertson. Accent: a communication-oriented network operating system kernel. Dans : *Proc. 8th Symp. on Operating Syst. Principles*, pages 64–75, Asilomar Conference Grounds, Pacific Grove CA (USA), décembre 1981.

[Shapiro 86] Marc Shapiro. Structure and encapsulation in distributed systems: the proxy principle. Dans : *Proc. 6th Int. Conf. on Dist. Comp. Sys.*, pages 198–204, IEEE, Boston MA (USA), mai 1986.

[Schantz 86] Richard E. Schantz, Robert H. Thomas, Girome Bono. The architecture of the Cronus distributed operating system. Dans :

Proc. 6th Intl. Conf. on Distributed Computing Systems, pages 250–259, IEEE, Cambridge, Mass. (USA), mai 1986.

[Stroustrup 85] Bjarne Stroustrup. *The C++ Programming Language.* Addison Wesley, 1985.

[Zimmermann 84] Hubert Zimmermann, Marc Guillemont, Gérard Morisset, Jean-Serge Banino. *Chorus: a Communication and Processing Architecture for Distributed Systems.* Rapport de Recherche 328, Institut National de la Recherche en Informatique et Automatique, Rocquencourt (France), septembre 1984.

SOS: a distributed Object-Oriented Operating System*

Marc Shapiro
Institut National de Recherche en Informatique et Automatique
B.P. 105, 78153 Le Chesnay Cédex, France
uucp: mcvax!inria!shapiro

Abstract

We present SOS, the operating system for SOMIW (Secure Open Multimedia Integrated Workstation). The SOS is object-structured; all services and communication is expressed in terms of passive objects with procedural interfaces.

Resources are accessed by clients via a local (to the client) *proxy* object. The possibly distributed or group nature of the resource is hidden behind the proxy interface.

The service and its representatives form a single distributed object (or *group*); any communication protocol is hidden, as part of its internal representation. The proxy is a programmable capability and may be different per-client. This approach also allows the programmer of a resource to cope with the heterogeneous nature of its clients.

The OS is based on a kernel which implements contexts, processes, memory segments and object descriptors. Object invocation normally occurs within one context, but may trap transparently into an other context.

1 Introduction

SOMIW (Secure Open Multimedia Integrated Workstation) is a co-operative project in building an advanced multi-media Office environment. The SOMIW Operating System (SOS) should facilitate the integration of software and hardware from different partners by implementing the object abstraction. SOS is an open operating system [LS79]; most of its functionality is flexibly provided outside the kernel, by system objects. Often, open operating systems [Che84,RR81,ZGMB84] place the emphasis on flexibility, and provide a low-level, uniform access to all distributed resources. They are based on a flat, network-wide address space and on a single simple communication primitive. We believe that this approach has reached its limits. There is a need to hide the internal structure of complex, distributed objects. There is a need for protocols more complex than a single-shot message or RPC. Uniformity gets in the way of applications which need to be aware of their host's and/or client's identity, and is an obstacle to supporting heterogeneous machines effectively. This is an other instance of the "end-to-end argument" [Sal81].

*This paper was presented at the 2nd European SIGOPS Workshop, on "Making Distributed Sytems Work", Amsterdam (The Netherlands), Sept. 1986.

2 SOS basics

2.1 An object-oriented system

An object is an instance of a class; there is a hierarchy of classes derived from each other. For the SOS, an object is simply a (collection of) data segment(s) connected to a table of procedures, itself pointing into a code segment. The SOS itself is a collection of system objects, running on top of an object kernel. An Object is passive; it is activated by *invoking* one of the procedures in its table. (This allows to use the type-checking C++ [Str85] compiler for building objects.)

In contrast to many existing systems, we offer support for the encapsulation of a complex, distributed resource behind a single black-box interface. This is done by forcing all communication between a client and a resource to go through a proxy, a single object representing the entire resource and local to the client.

2.2 The kernel

The kernel objects are: contexts, processes, segments, and acquaintance descriptors.

Contexts are independent virtual memory spaces. An object mapped into a given context (i.e. visible from within it) is called an acquaintance of that context.

Acquaintances are known to the Kernel via acquaintance descriptors. An A.D. contains: a pointer to the first data segment of the acquaintance; a pointer to its method table (table of procedures); and possibly a `trapReference` pointing to an object of the same group in another context (explained below). Otherwise the kernel knows nothing of the structure of the acquaintance's data, which is compiled into the code for the acquaintance.

Any number of processes may execute concurrently within a context.

2.3 Communication

The basic form of communication is the invocation of some procedure of an acquaintance; this is a simple, local procedure call (indirectly through the method table).

For communication with another context, a "trapdoor" proxy is needed. A typical scenario is the following: object A in context C1 is a proxy for a resource implemented by object B of context C2. Some procedures of A contain code to be executed on its local data; these are executed by the normal intra-context invocation. Others may have to perform a function on the data of B. Say `A.foo()` is one of those procedures; its method table entry is filled with a pointer to a `crossInvoke()` (a kernel trap) and the `trapReference` in its Acquaintance Descriptor points to object B in context C2. An invocation of `A.foo()` causes the process to continue execution in context C2, invoking `B.foo()`. When it terminates, control returns in C1.

Remote communication is not supported directly but is possible by arranging for the `trapReference` to point to a communication object with access to the network devices. This implements a sort of a virtual circuit, with a peer communication object on the remote workstation (or with a set of peers on any number of machines). The default communication object simply extends cross-context invocation to Remote Procedure Call on a reliable connection; more efficient, application-oriented protocols may be used easily since the protocol remains internal to the group. In fact, the SOS restricts communication patterns to enforce this group consistency rule.

3 The Proxy Principle

The proxy principle states that, in order to perform operations on some resource, a client must invoke operations of a proxy, an acquaintance of the client's context representing the resource.

A request to the Acquaintance Service to import a named proxy (data and code) allows to gain access to new resources. The Proxy is created dynamically by the service, exported, and instantiated in the client's context.

The proxy filters the client's requests; it may either reply if there is enough information locally to do so; or signal an exception; or forward the request to its principal.

The proxy principle is a tool for encapsulation, the implementation of the resource being hidden behind the proxy interface. The programmer for the resource is encouraged to put as much functionality as possible into the proxy running in the caller's workstation, promoting performance transparency.

The proxy and its principal(s) collectively implement one distributed object or group. The resource may be implemented as a complex network of communicating objects. The protocol between them is part of its internal representation and invisible to clients.

4 Dependent Objects

In the SOS, the dynamic relations between objects are explicit. A relationship between two contexts (on the same workstation) is indicated by the existence of a trapdoor between them. A relationship between two workstations is indicated by the protocol object connecting them. This information allows the SOS to maintain a dependency graph for objects. Thus it is possible to broadcast a signal to all your dependents. Furthermore, the destruction of an object is automatically signalled to its dependents. This signal travels along the same paths as normal communication, and is correctly ordered w.r.t. the latter.

The primary use of this feature is for objects managing system information (e.g. locks) outside of the kernel to release unused resources.

5 Current state and problems

The functional specifications for most of the system have been written. We are currently porting and adapting the Chorus distributed kernel [ZGMB84] to meet our needs. We have also done some simulation of the dynamic importation of an object into a context,

and of the dynamic linking of the code. Completion of the SOS is planned for the end of 1987.

Dynamic linking introduces new problems. It must be checked at object importation time that the code being imported is compatible with that already present. This implies modifications to our C++ production chain, which will be implemented at a later date.

A standing goal of the project is to allow mixing of different activity types within a running system, from real-time activities (voice and image) to transactions and highly-available activities (reliable office procedures). Protocol objects will ensure that different activity types do not conflict, and that each activity pays only the price of its own mechanisms. Preliminary investigation suggests that the context/client/proxy mechanism alone is too static for an adequate support of generalized activities. We are currently investigating a more dynamic mechanism based on a particular utilization of virtual memory.

References

[Che84] David R. Cheriton. The V-Kernel, a software base for distributed systems. *IEEE Software*, April 1984.

[LS79] B. Lampson and R. F. Sproull. An open operating system for a single-user machine. In *Proc. 7th Symp. on O.S. Principles*, pages 98–105, 1979.

[RR81] R. Rashid and G. Robertson. Accent: a communication-oriented network operating system kernel. In *Proc. 8th Symp. on Operating Syst. Principles*, Asilomar Conference Grounds, Pacific Grove CA (USA), December 1981.

[Sal81] J. H. Saltzer. End-to-end arguments in system design. In *Proc. 2nd. Int. Conf. on Distributed Computing Syst.*, pages 509–512, Versailles (France), April 1981.

[Str85] Bjarne Stroustrup. *The C++ Programming Language*. Addison Wesley, 1985.

[ZGMB84] Hubert Zimmermann, Marc Guillemont, Gérard Morisset, and Jean-Serge Banino. *Chorus: a Communication and Processing Architecture for Distributed Systems*. Rapport de Recherche 328, Institut National de Recherche en Informatique et Automatique, Rocquencourt (France), September 1984.

A dynamic link editor for C++

Philippe Gautron
Marc Shapiro
Institut National de Recherche en Informatique et Automatique
B.P. 105, 78153 Le Chesnay Cédex, France
uucp: mcvax!inria!corto!gautron

Abstract

We present the design and implementation of a dynamic loader and linker for the C++ language. The motivations and goals of this tool are discussed, as well as its introduction into a C++ compilation chain. A keyword was added the C++ language, and a few modifications to the compiler were necessary. The implementation is clean and portable. We present the rationale of this work, the necessary changes to the language, and a working UNIX implementation.

1 Introduction

The conventional production cycle for a C or C++ program is a sequence of edit–compile–link–load–run phases. Each source file (".c" files on a Unix system) is compiled, producing an equal number of binary relocatable files (".o"). These are then linked together and with all necessary libraries (".a" files), producing a single, non-relocatable binary program image ("a.out"). The latter may be executed any number of times with identical results.

Compilation before execution is characteristic of compiled languages. Linkage before execution, which we will call static linking, is often imposed by the programming environment, and is not always desirable. Dynamic linking, i.e. delaying the binding between procedure names and their code until execution time yields a number of benefits:

- Program images are smaller. With static linking, all program images must be linked in advance with all the procedures which they may ever have to call. Each identical copy of the code for library procedures takes up space on disk, which may be excessively large. With dynamic linking, procedures are read in at run time as needed, and therefore do not take up space in the program image.
- For similar reasons, the production cycle time is shorter. Static linking with a large library can take many minutes.
- Shared libraries are easier to implement. When linking is static, each executable image has its own copy of common procedures, and it would be very hard for the loader to recognize that they are in fact identical. With a dynamic loader/linker this is not the case.
- Greater flexibility. If one wishes to fix a bug or change the implementation of a library, static linking imposes to re-link all the program images. For a large system, this is a non-trivial task. With dynamic linking instead, each execution is guaranteed to use the most recent version of all the procedures.

The main advantage of static over dynamic linking is reduced execution time: a dynamically-linked program incurs the overhead of linking at each execution, whereas static linking is done only once. For this reason, it is desirable to allow a single program to combine both static or dynamic linking, as necessary.

Dynamic linking is a natural benefit of interpreted languages. Dynamic linking may be added to compiled languages also, if there is sufficient information in the binary file to: (1) detect that a new procedure needs to be imported; (2) extract the symbolic name (and possibly, type information) of that procedure; (3) discover the file containing the procedure's code.

In the Multics operating system [Org72], the program loader performs dynamic linking for any program in any language. The Unix[1] loader exec(2) has also been modified to yield similar benefits [Arn86]. An attractive alternative is an implementation which does not modify the kernel. For instance, the Andrew project [MSC*86] uses "Camphor," a dynamic linking system for C programs. Camphor consists of a set of preprocessor macros, support programs, and run-time li-

[1] UNIX is a trademark of the Bell Laboratories.

braries (linked statically). An implementation outside the kernel cannot support shared libraries.

We present here a dynamic linker for the C++ object-oriented programming language [Str85]. Our work avoids pre-processor macros by integrating linker support into the compiler. This necessitates a small addition to the syntax of C++, and some additions to its code generator.

We link in the code for a class at the time of its first instantiation. We can also link external procedures (i.e. which are not class members), with some restrictions.

Our implementation builds upon the Camphor implementation. Like Camphor, we keep a static link phase, and do not support shared libraries. Our implementation is clean and machine-independent. It depends only on the format of binary files (`a.out.h`; a COFF version is contemplated).

This paper distinguishes between the *exported* relocatable modules, and the *importer*, i.e. the program into which these modules are linked at run time.[2] Each of these has both a source and a binary image. The use of our dynamic linking system may be divided into four independent steps:

- *definition* of exported procedures by the exporter
- *declaration* of imported procedures by the importer
- *detection* that the importer needs to load a new module, by the linking system
- *loading and link-editing* the module into the importer.

The first two steps are performed by the compiler, using information provided by a small additional syntax. The detection step is performed at run-time by code generated by the compiler. The last step is performed by a run-time support library.[3]

The work described here is part of SOS, a distributed object-oriented operating system for an integrated office-automation environment [Sha86]. In SOS, objects are routinely migrated from one machine and/or one address space to another. The code for the object is migrated along with its data, and dynamically linked into the receiving address space. A description of SOS is outside of the scope of this paper, which is restricted to the description of our dynamic-linking C++ environment on Unix 4.2BSD.

We will now describe our modifications to the C++ language and compiler. Then we describe our implementation of detection, loading and linking, based on the corresponding Camphor implementation.

[2]This distinction is made essentially for the sake of clarity of exposition, as rôles shift during execution. After linkage, an exported module becomes part of the importer, and its execution may cause further importations.

[3]Which is linked statically to the importer.

2 Dynamic linking in the C++ compiler

To ease the use of dynamic linking, we have added a small amount of syntax to C++ [Str85], and done some additions to the compiler's code generator. C++ is an extension of C [KR78]; a C++ "class" (a generalization of the C "struct") is an abstract data type: its data is encapsulated with the procedures, or *methods*, which are allowed to read or write that data. Our extension adds the new keyword `dynamic`. A `dynamic class` is a class for which the code of methods is "dynamic", i.e. either imported (for declarations) or exported (for definitions).[4] The code generated for calls to dynamic procedures uses an indirection through a pointer to the procedure. This pointer initially points to an error procedure. The modified compiler detects when the methods are actually needed; then a (statically-linked) procedure will be called at run time, to load in its code, resolve addresses, and fill the pointer with a legal value.

The distributed C++ compiler[5] actually is a front-end, generating C code. We will present, both our design, and the actual modifications to the generated code.

3 C++ classes

We will first present the unmodified C++ class mechanism, and the corresponding generated code.

A class is an abstract data type [Gau84]. C++ objects are instances of some class; each instance is a black box. Syntactically, a class resembles a C structure: named data fields or *members* define its internal representation. In addition, a class defines the procedures or *methods* which are allowed to read or write the representation. A *constructor* is a distinguished method which is automatically called when an instance is created (for initialization). Similarly, a *destructor* is automatically called when an instance is destroyed. For some class `foo`, the constructors are also called `foo` (there may be zero, one or many constructors, distinguished by their argument types). The destructor is called `~foo`.

A class may be *derived* from some other, *base*, class. The derived class inherits the representation and the

[4]Optionally, some methods may be dynamic, others statically linked, and others "inline".

[5]Cfront version 1.1.

methods of its base, and may add more fields or methods. In addition, a method declared *virtual* in the base class may be redefined in the derived class. Whereas (in unmodified C++) non-virtual methods are called directly by name, virtual methods are called indirectly via a *method table* attached to an instance; the address of this table is stored in a field of each instance.

Consider the following class declaration:

```
class foo {
   int field;           /* private data       */
 public:                /* only what follows
                           is visible:       */
   foo ();              /* a constructor with
                           no arguments      */
   ~foo ();             /* the destructor    */
   int meth1(int);      /* an ordinary method */
   void meth2 ();       /* an other          */
   virtual int virt(); /* a virtual method   */
};
```

The above declaration must be completed by appropriate definitions for the methods, e.g.:

```
foo::foo () { field = 0; }
foo::~foo () { printf("foo destructor\n");}
int foo::meth1 (int i)
  {int j; j=field; field=i; return j;}
void foo::meth2 () { field++;}
virtual int virt() { return field; }
```

The notation `int foo::meth1 (int i)` reads: "method `meth1` for class `foo`, which takes a single `int` argument and returns an `int`." The syntax of method invocation, for some object F of class `foo`, is:

```
  {foo F;                /* declare */
   ...
   x = F.meth1(1024);    /* invoke  */
   y = F.virt();         /* invoke  */
   ...}
```

All methods have an implicit first argument, a pointer to the object. A constructor or a destructor cannot be called explicitly: they are called by the compiler when an object is created or destroyed.

The C code corresponding to the declaration of class foo, as generated by the front-end, will be (after simplification for legibility):

```
struct foo {           /* declaration */
   int field;
   int (** vptr) ();  /* ptr to method table */
   };

int (*foo__vtbl[]) = { /* meth. table for foo */
   (int(*)())virt,     /* only 1 virt. method */
   0
}
```

```
_foo__dtor (this)     /* body of destructor */
   struct foo* this; {
       printf ("foo destructor\n")
}
int _foo_meth1 (this, i) /* foo::meth1 */
   struct foo* this; int i; {
       int j;
        j=this->field; this->field=i;
       return j;
}
void _foo_meth2 (this)   /* foo::meth2 */
   struct foo* this; {
       this->field++;
}
int _foo_virt (this)     /* foo::virt */
   struct foo* this; {
       return this->field;
}
```

The generated constructor contains, in addition to the user code, some code generated by the compiler to initialize the pointer to the method table:[6]

```
_foo__ctor (this)      /* body of constructor */
   struct foo* this; {
     /* generated by compiler: */
   this -> vptr = foo__vtbl;
     /* user-supplied code: */
   this -> field = 0;
   }
```

When an instance F of class `foo` is allocated, for instance on the stack, with a declaration like:

```
{
    foo F;
    ... use F ...
}
```

the front-end generates the following C code:

```
{
   struct foo F;     /* allocate instance */
   _foo__ctor (&F);  /* initialize it     */
   ... use F ...
   _foo__dtor (&F);  /* destroy it        */
}
```

When an ordinary method is invoked, the front-end generates a simple C procedure call by the method's name. When a virtual method is called, instead, the call is indirect via the method table, so that it can be replaced easily in derived classes:

```
   x = F.meth1 (1024);
   y = F.virt();
```

[6] A method table, this pointer, and its initialization code, are generated only for classes containing at least one virtual method.

generates the following C code (note the implicit first argument):

```
x = _foo_meth1 (&F, 1024);
y = (*F.vptr[0])(&F);
```

4 Declaration, detection, and invocation of a dynamic class

When a class is declared dynamic, the compiler must generate sufficient information for its methods to be loaded, linked and then called dynamically. Therefore, a call to a method of a dynamic class is made to occur indirectly through a method table, the address of which is stored in a field of the instance's data.[7] At dynamic load time, the effective address of the procedure is computed and inserted into the table, thereby accomplishing the dynamic link (see section 5.2).

For example, if dynamic is added to the declaration of class foo:

```
dynamic class foo {
    ...
};
```

then our modified front-end generates the following C code for the declaration:

```
struct foo {
   int field;  /* data */
   int (** vptr)();  /* virt. meth. tbl. */
   int (**dmptr)();  /* ord. meth. table */
   int (**ddptr)();  /* destructor table */
};
```

A method invocation generates indirect calls via one of the above tables. For instance, invoking method meth1 on the object F of class foo generates:

```
x = (* F.dmptr[0]) (&F, 1024);
```

4.1 In the exporter

The initialized method tables are created for the exporter only:

```
int (*foo__vtbl[]) = {     /* virtuals     */
   (int(*)()) virt, 0
};
int (*foo__dmtbl[]) = {    /* ord. methods */
   (int(*)()) meth1,
   (int(*)()) meth2, 0
};
int (*foo__dctbl[]) = {    /* constructors */
   (int(*)foo__ctor, 0
};
int (*foo__ddtbl[]) = {    /* destructor   */
   (int(*)foo__dtor, 0
};
```

The virtual method table is as before. The tables for ordinary methods, constructors, and for the destructor, are built along the same lines.[8]

In the exporter, the code of constructors is again supplemented with instructions to initialize the pointers to the method tables:

```
_foo__ctor (this)
   struct foo* this; {
     /* compiler-generated code: */
   this -> vptr = foo__vtbl;   /* as before  */
   this -> dmptr = foo__dmtbl; /* ord. meth. */
   this -> ddptr = foo__ddtbl; /* destructor */
     /* user-supplied code: */
   this -> field = 0;
}
```

4.2 In the importer

The code generated for a call to a method of a dynamic class, whether ordinary, virtual, or the destructor, is an indirect call via the corresponding table, which will be pointed to by a field of the instance. For constructors, however, there is no per-instance table; constructor invocation is via a global, per-class table, which must be generated by the compiler. In the importer, this table is initialized with an error function:

```
/* global constructor table */
extern int (*foo__dctbl[]) = {
   (int(*)()) dynamic_link_trap; 0 };
```

The actual importation of code is delayed until its first use is detected. For dynamic classes, detection always occurs when executing a constructor. Therefore, the modified compiler generates detection code in the importer for every instantiation of a dynamic class. Thus, the declaration:

```
dynamic foo F;
```

generates the following C code, allocating space for the instance, invoking the dynamic loader/linker, then invoking the constructor via the constructor indirection table:

```
struct foo F;                    /* allocate space
*/
dynamic_link (&foo__dctbl, 0); /* import + link
*/
```

[7] Just like virtual methods in unmodified C++. The exceptions are "inline" methods and constructors. The former are expanded inline. The latter are not operations on an instance, but on a class. They are called indirectly via a per-class constructor table.

[8] It would have been possible to consolidate the tables for virtual, ordinary, and destructor methods into a single table, but it must be easy to find which one is the destructor. It is desirable to keep a separate table for constructors.

```
(*foo__dctbl[0])(&F);          /* constructor
*/
```

As we have seen before (see section 4.1), the execution of the constructor will fill in the indirection tables for ordinary, virtual and destructor methods.

5 Implementation of dynamic load and linking

5.1 Preparation of the exported code

The work of the dynamic loader/linker `dynamic_link` is twofold: first, the exported code is read into the importer, and its relocatable data is updated to reflect its actual location. Second, the importer's constructor table is filled with the actual location of the constructors. This is made possible by a special preparation of the exporter: a distinguished procedure is generated, which will return the resolved addresses of the constructors, when called after relocation.

For each source file `bar.c` containing the definition of dynamic class `foo`, the compiler generates a single procedure, called `_dlinit_bar_c_` which will return the address of the constructor table:

```
static int (*foo__dctbl[2])() = {
    (int(**)()) _foo__ctor,
    (int(**)()) 0
};

int (** _dlinit_bar_c_ ()) () {
    return foo__dctbl;
}
```

This procedure is made distinguished, and easily available to the dynamic linker, by declaring it to be the "entry procedure" of the relocatable binary file `foo.o`, by using the appropriate options of the Unix static linker:[9]

```
cc -c foo.c
ld -e __dlinit_foo_c_ foo.o -o foo.e
```

5.2 Dynamic Load

A call to the `dynamic_link` procedure is inserted by the compiler at each instantiation of a dynamic class. It is the responsiblity of `dynamic_link` to:

- keep track if the code for class `foo` is already loaded or not
- if not already loaded, find the corresponding binary file, load it, and bind unresolved references.
- fill in the constructor table `foo__dctbl` with resolved constructor references.

The call to the procedure `dynamic_link` causes the exported binary file to be loaded into the importer's data.[10] The relocatable data contained in the file is updated at this time. The addresses of the constructors defined in the file are known by executing its entry procedure (see section 5.1). The constructor table, passed as an argument of the `dynamic_link` call, is filled with these addresses. This terminates dynamic linking. The other tables (to ordinary, virtual and destructor methods) are not filled by `dynamic_link`, as they will be filled as a side-effect of calling a constructor.

Note that the above is different from the strategy used in Camphor (see appendix A.1) and similar implementations, where dynamic linking is a side-effect of the first call to a dynamic procedure. Our strategy involves a slight overhead, but it is more portable and more extensible. We could change easily to the Camphor strategy by making `dynamic_link` a null procedure, and performing the above algorithm in `dynamic_link_trap` instead.

5.3 External procedures

Calls to external procedures from within the exporter must also be linked dynamically. We chose not to generate indirect calls in this case, but to keep direct calls.

All external procedures called by any eventual imported code must be statically linked with the importer, and described by a table of the the following structure:

```
struct library{
    char* name;     /* function name */
    int (*ptr)();   /* function ptr  */
};
```

For example, the description of the C library contains:

```
struct library libc[] = {
    "_printf", printf,
    ...
};
```

When an exported file is imported, the dynamic linker resolves references to external procedures by searching the above-described tables, which are known at static link time.

This solution is not very satisfactory and doesn't extend well. We are currently exploring alternatives.

[9] Any binary file may have an entry procedure. The normal use of this is to distinguish the main procedure of a program.

[10] The name of the file may be explicitly given by optional arguments to the `dynamic` declaration, as in:
`dynamic ("/exports/foo.e") foo F;`
Such filenames are passed as extra arguments to `dynamic_link`.

6 Conclusion

Benchmarks on the different uses of class methods (simple, inline or virtual) and communicated by B. Stroustrup showed that the overhead of indirect calls to virtual methods is negligeable. Our implementation generalizes indirect calls to all methods of a dynamic class.

A first implementation currently runs on Sun-3 and Vax running BSD Unix. It is portable as long as the C++ compiler is. The only dependency is the format of the loader `a.out.h`. A version for Unix System V COFF format is planned.

The rules for finding imported files are specified in a Unix "environment variable", DLPATH.

The current version does no name or type-checking. There is no assurance that the imported file is the right one. The importer's method tables are assigned what is returned by the imported entry procedure, without any checking of sizes or contents. Clearly this is undesirable. Our next developments will be to implement a full name- and type-checking at importation time, with minimal overhead. This development should also allow us to use shared libraries.

Acknowledgments

Our acknowledgments to the different authors of Camphor: P. Smith, B.Sacks, S. Daniel, M.Kazar, D. Nichols, A. Palay, F. Hansen ; to the C++ author, B. Stroustrup ; to our colleagues of the SOMIW project: V. Abrossimov, S. Habert, M. Makpangou , L. Mosseri; and to R. Ehrlich et J.D. Fekete.

Appendix A: The Camphor implementation

This appendix is a brief, and considerably simplified, exposition of the Camphor dynamic linking environment, upon which we have built. Camphor is a set of preprocessor macros (which generate the indirect calls, the entry procedures, and the tables described hereafter) and a dynamic loader and linker, which is linked statically with the importer. Exported files are prepared specially, as described below.

A.1 Invocation and detection

Calls to dynamic procedures are indirect via a structure of the following type:

```
struct Link {
   int (*ptr)(); /* pointer to procedure */
   char* name;   /* name of procedure    */
};
```

The pointer initially points to an illegal instruction. The first time a dynamic procedure is called, execution of the illegal instruction raises a Unix signal; the handler for this signal is the dynamic loader/linker. The latter finds the address where the illegal instruction occured from the signal arguments; the name of the procedure is taken from the next field in the `Link` structure. The code is then imported (read from a file, the name of which is deduced from the procedure name). The pointer is then changed to reflect the actual procedure address (see next paragraph), and the procedure call continues. The second time the same procedure is called, the pointer already has the correct value.

The Camphor method of invocation and detection is cost-effective, but not very portable. It relies on non-standard peculiarities of their C compiler, on some assembler code, and on the implementation of signals in Unix 4.2 BSD.

A.2 Dynamic linking and loading

The exporter is a binary file, specially prepared. Like any binary file in the Unix environment [KP84], it has a distinguished procedure called the entry procedure. This is set to a macro-generated procedure, which returns the address of a table of `Link` structures, describing the exported procedures. At dynamic link time, the effective address of an imported procedure is found by invoking the entry procedure, which returns the table, and searching through this table.

The Camphor method for loading is quite effective, and is used almost verbatim in our implementation.

References

[Arn86] James Q. Arnold. Shared libraries on UNIX system V. In *Summer Usenix Conference*, Atlanta (USA), June 1986.

[Gau84] Philippe Gautron. Instanciation, héritage et transmission de messages en C++. In *Actes des deuxièmes journées d'études sur les langages orientés objets*, AFCET, Brest (France), November 1984.

[KP84] B.W. Kerninghan and Rob Pike. *The UNIX Programming Environment.* Prentice-Hall, 1984.

[KR78] B. W. Kerninghan and D. M. Ritchie. *The C Programming Language.* Prentice Hall, 1978.

[MSC*86] J. H. Morris, M. Satyanarayanan, M. H. Conner, J. H. Howard, D. S. H. Rosental, and F. D.

Smith. Andrew: a distributed personal computing environment. *Communications of the ACM*, 29(3):184–201, March 1986.

[Org72] E.I. Organick. *The Multics system: an examination of its structure*. MIT Press, Cambridge, Mass. (USA), 1972.

[Sha86] Marc Shapiro. SOS: a distributed object-oriented operating system. In *2nd ACM SIGOPS European Workshop, on "Making Distributed Systems Work"*, Amsterdam (the Netherlands), September 1986. (Position paper).

[Str85] Bjarne Stroustrup. *The C++ Programming Language*. Addison Wesley, 1985.

On the use of the dynamic link editor

Philippe Gautron
Marc Shapiro
Sabine Habert
Institut National de Recherche en Informatique et Automatique
B.P. 105, 78153 Le Chesnay Cédex, France
uucp: mcvax!inria!corto!gautron

1 Introduction

This document presents the use of the dynamic linker, both for Unix and for SOS. The rationale and implementation of the C++ dynamic linker are explained in another paper [GS87]; this one simply describes its use, and possible problems.

This document describes our current implementation (Prototype Version 1), running on a Sun-3 release 3.0 or 3.2.

The dynamic linker is in two parts: one (modified code generation) is integrated in the C++ compiler distributed as part of "SOS Prototype V1" (in directory `~sos/v1/bin`). The other (dynamic search and linking itself) is encapsulated in the procedure `dynamic_link`, which has two versions: one for Unix, and one for the SOS Prototype V1. Only the latter is distributed herein (in directory `~sos/v1/lib`).

The C++ grammar has been modified by the sole introduction of the keyword `dynamic`.[1] The usage of the dynamic linker is for the most part transparent, being automatically supported by the modified compiler and by `dynamic_link`. However, users must distinguish between:

- The importer (of some exported code compiled elsewhere); the client in SOS.
- (In SOS) the principal, which exports dynamic instances to clients, but is not itself dynamic.
- The exported code (to be dynamically linked into the importer).

Exported code must be specially prepared, by compiling it with the shell command script `makexport`.

Exportation is intimately bound to the class concept. The keyword `dynamic` may be applied only to a whole class.

[1] And not `dynalink` as indicated incorrectly in some versions of [GS87].

2 Declaration of a dynamic class

A typical declaration of a dynamic class, FOO, follows:

```
dynamic class FOO{
    public:
        int champ;
        FOO();
        ~FOO();
        int m1();
        virtual int m2();
};
```

The only difference with a classical class is the addition of the keyword `dynamic`.

It is best to put this declaration in a ".h" file, which will be included both by the exporter and by the importer. Declarations must be consistent in the importer and the exporter.

BEWARE Checking this consistency is up to the users. There is currently no automatic way to check this. If the declarations are not cosnsitent it may cause wierd, hard-to-debug errors.

This checking will hopefully be made automatic in future versions.

2.1 Export or proxy code

The exporter module (the proxy in SOS terminology) *defines* the code of the dynamic class. This module should be compiled using the command `makexport`. The syntax of this command is:

```
makexport [ [+z className] [-e name] options ]
file.c
```

This command produces a special binary file, named `file.e` by default.

Separate compilation of dynamic classes is not supported: all the code for a dynamic class must be in

the same file. Only code of one dynamic class must be *defined* per file. If several dynamic classes are declared in the file, the option `+z className` must be given to `makexport` to indicate the one class defined therein.

The "-e" option allows to re-name the produced file. Any other options are transmitted verbatim to the C++ compiler or the loader.

2.2 Import code

Importation is bound with instanciation. Every instantiation of a dynamic class automatically generates a call to the dynamic linker/loader procedure `dynamic_link`.[2]

Two typical declarations of an instance of a dynamic class follow:

```
const char* searchName;
/* auto instance of the block */
dynamic (searchName) FOO instance1;

/* instance on heap */
dynamic (searchName) FOO* instance2_ptr;
...
instance2_ptr = new FOO;
```

BEWARE The value `searchName` is evaluated at the instanciation time. To avoid problems, make sure it is either a compilation constant or declared `const char*`.

When using the first instanciation style, a call to `dynamic_link` will occur preceding any instruction in the block.

In the second case, `dynamic_link` will be invoked when `new` is executed.

BEWARE It is possible for a single class to have both dynamic and non-dynamic instances, but not in the same source file. The procedure `dynamic_link` is called only for those instances which are declared `dynamic`.

2.3 Compilation options

An importer must be compiled by calling CC (the C++ compiler) with the option `-DL` (for "dynamic link"), which causes the "dynamic link" version to be used and the appropriate libraries to be linked.

The compiler may also be called with the option `+Z` which has the effect of ignoring any `dynamic` declarations (i.e. all code is statically linked).

[2] The procedure `dynamic_link` is written such that only the first effective instanciation of a dynamic class will effectively load in its code. For subsequent instantiations of the same dynamic class, `dynamic_link` recognizes that the code is already there.

SOS programs are compiled using the `sosCC` command. In SOS, a principal, which wants to create and export proxies must be compiled with the `+Z` option. This allows the principal to declare proxy objects non-dynamic.[3]

The code for a proxy (exported code) must be compiled using the `makexport` command.

3 Scenario of an instantiation

Two typical instantiations for a dynamic class are like this:

```
dynamic (searchName) className variable1;
dynamic (searchName) className * ptr_variable2;
...
ptr_variable2 = new className;
```

which declares "variable1" and "variable2" to be of dynamic class "className". The scenario of an instantiation, as generated by the modified compiler, is the following:

1. call to the dynamic loader/linker (function `dynamic_link`); a search is performed.

   BEWARE The effect of this search is very different in Unix and in SOS.

   In Unix, the character string `searchName` is simply the name of a file where the code for class "className" may be found; the object is *instantiated* by calling its constructor. In SOS `searchName` is the symbolic name of a principal. The principal is asked to export a proxy, which is then *imported*. See below.

2. call to the class constructor, as in unmodified C++

3. call to an initialization function `sosObject::initialize`. This also has different effects in SOS and in Unix.

None of the usual C++ mechanism is changed. In particular the instantiation mechanism for non-dynamic classes or variables remains the same as before. Note that inline constructors or member functions are expansed at compilation time, and therefore are not loaded dynamically.

[3] Proxy objects are first created locally by the principal, then exported to clients. Clients import proxy data and code dynamicly; principals do not.

3.1 In Unix

The code for the object is found, if needed, in a file; the name of the file is the `searchName` string.

BEWARE This means that only the `searchName` of the instance which occurs the first in time is significant.

By default, if `searchName` is not specified, `className.e` is used; for instance, the following declarations of `f1` and `f2` are identical:

```
dynamic FOO f1;
dynamic ("FOO.e") FOO f2;
```

This file is found in one of the directories named by the UNIX environment variable `DLPATH` (similar to the variable `PATH`). For example, to import code held in the directories "/export1" and "/export2", do, using the Bourne or the Korn shell:

```
DLPATH="/export1:/export2"; export DLPATH
```

or, using the C shell:

```
setenv DLPATH "/export1:/export2"
```

DLPATH is searched from left to right in the declaration order.

3.2 In SOS

In SOS, an instance declared `dynamic` is assumed to be a proxy and the declarer is assumed to be its client. Instanciating a dynamic object triggers an importation. `SearchName` is the symbolic name of a principal, registered with the Name Service. The principal is asked for a proxy, which is then imported, becoming the new instance.

Consider for instance an instance XI of dynamic class XC:

```
/* file client.c */
const char searchName[] = "somePrincipal";
dynamic (searchName) XC *XI_ptr;
. . .
XI_ptr = new XC; // import proxy for
                 // "somePrincipal"
```

The actual scenario is the following:

1. Before anything, the principal must register its name with the Name Service, and create a `code` object to hold the code of class XC. The code is loaded from a Unix file; the name of this file is declared in the `code` constructor, and evaluated relative to `DLPATH`, as in section 3.1.

   At the time of the instanciation of XI:

2. The Acquaintance Service queries the Name Service for the principal, using the `searchName` `"somePrincipal"`.

3. The principal's `giveProxy` procedure is called. This terminates:

   - either with a `refused` exception (which is resignaled to the client). In this case, the following steps are skipped.
   - or by normal termination, declaring a proxy and its associated code object.

4. The proxy data and (if necessary) its code object are imported into the client's context.

5. XI is instantiated by calling its constructor (but see "Beware" below).

6. `((sosObject)XI).initialize()` is called, which overwrites XI with the imported data.

   BEWARE No matter what class is declared for `XI`, it is always `sosObject::initialize` which is called. Any prior initialization performed by the constructor is always overwritten by the imported data.

4 Example

```
$ DLPATH="/export"; export DLPATH        #
Bourne shell "sh"
$ cat essai.h
dynamic class FOO{
 public:
    FOO();
    int m();
};

$ cd /export
$ cat exporter.c

#include "essai.h"

 FOO::FOO() {
    printf( "constructor reached\n" );
}

 FOO::m() {
    return printf( "method m reached\n" );
}

$ makexport -e essai.e exporter.c
$ cat importer.c

#include "essai.h"

main(){
    dynamic ("essai.e") FOO instance;
    instance.m();
}
```

```
$ ls
essai.e                 essai.h
exporter.c      importer.c

$ CC -DL importer.c
$ a.out
constructor reached
method m reached
```

5 Conclusion

The goal of this first version was feasibility study. This goal is reached, many problems remain.

References

[GS87] Philippe Gautron and Marc Shapiro. A dynamic link editor for C++. January 1987.

Exception Handling in C++ Programs.

Vadim Abrossimov

1 Exception Handling

The exception handling mechanism consists of two parts:

- *raising* of exceptions
- *handling* of exceptions.

Raising is the way a callee function notifies its caller of an exception condition; handling is the way the caller responds to such notification. A raised exception always goes to the *immediate caller*, and the exception *must* be handled in that caller. When an exception is raised, control is immediately transferred to the closest applicable caller's *handler*. If the caller not handles that exception, an *exception failure* error occurs.

2 Exception Raising: Raise Statement

An exception is raised with the `raise` statement, which has the form:

raise-statement:
`raise(` *exception-name* `);`

A `raise` statement may appear anywhere in the body of a function. The execution of the statement begins with destruction of local objects. The function is then terminated and execution continues in the caller, as described below.

3 Exception Handling: Exception Block

The *exception block* is a form of C++ *compound-statement*:

compound-statement:
statement-list$_{opt}$
exception-block

The exception block has the form:

exception-block:
`begin`
handled-block
handlers
`end`

where

handled-block:
statement-list$_{opt}$

handlers:
`except`
handler-list$_{opt}$

handler-list:
handler
handler handler-list

handler:
`when(` *exception-name* `)`
handler-body

handler-body:
statement-list$_{opt}$

exception-name:
identifer

`others`

The *handled block* is the block to which the *handlers* are attached. Each *handler* specifies one *exception name* and a *handler body*. The handler body is executed if an exception with target exception name is raised by an invocation in handled block. All of the names in a handler list must be distinct. The optional `others` clause is used to handle all exceptions not explicitly named. The handled block can contain any C++ statement, and can even be another exception block.

If, during the execution of handled block, some invocation raise an exception, control immediately transfers to the syntaticaly closest aplicable handler. When excecution of the handler completes, control passed to the statement following the handler's exception block. If excecution of the handled block completes without raising an exception, the attached handlers are not executed.

An exception raised inside a handler is treated the same as any other exception. [1]

4 Reraise Statement

During the excecution of a handler the exception which it handles can be *reraised*:

reraise-statement:
`reraise;`

5 Example

We now present an example demonstrating the use of exception handlers. We write a function, *stringToInt*, wich converts string to unsigned integer.

```
#include <sos/sos.h>
    unsigned
    int stringToInt( char* str )
{
    unsigned result= 0;
    while( *str ) {
        int i= *str++ - '0';
        if( i < 0 || i > 9 ) {
            raise( invalidCharacter );
        }
        unsigned newResult= result*10+i;
        if( newResult < result ) {
            raise( integerOverflow );
        }
        result= newResult;
    }
    return result;
}
```

Now we can use this function as follow:

```
...
unsigned r;
char* str;
...
begin
    r= stringToInt( str );
except
    when( invalidCharacter )
        ... // error processing
    when( integerOverflow )
        ... // error prcessing
end
...
```

6 Bugs

If a procedure, returning a non-void contains a `raise` or a `reraise` statement (like `stringToInt` in the example above), the compiler will complain about not returning a value. These error messages may safely be ignored.

[1] Note that an exception raised in some handler can not be handled by any handler attached to same handled block; either it handled within the handler, or it handled in embedded exception block.