Static vs. Dynamic CMIP/SNMP Network Management Using CORBA

Luca Deri
IBM Zurich Research Laboratory[1], University of Berne[2]

Bela Ban
IBM Zurich Research Laboratory[1], University of Zürich[3]

The increasing complexity and heterogeneity of modern networks is pushing industry and research to look for a single and consistent way of managing networks. With the advent of open object-oriented distributed computing models such as CORBA, there are efforts to make the operational and management models the same, i.e. to manage and operate the network using CORBA.

The aim of this paper is to show some techniques that allow to manage CMIP/SNMP network resources using CORBA. Static techniques which map each managed object class into a corresponding CORBA interface are compared with dynamic techniques which rely on runtime information. Finally this paper demonstrates that CORBA-based network management applications are becoming attractive in terms of efficiency and application size, overcoming limitations of early solutions.

Keywords: Network Management, CMIP, SNMP, CORBA, Scripting Language.

1. Introduction

With the growing impact of CORBA [OMG] in the telecommunications sector, the need has risen to employ CORBA to manage CMIP/SNMP agents. Since the CORBA object model (using IDL as specification language) is easier to learn than CMIP [CMIP] and SNMP [SNMP], anyone who is able to create CORBA applications can immediately use services offered by CMIP/SNMP agents, given a CORBA interface to them, without having to have a specific knowledge about CMIP or SNMP. It is the authors' opinion that this asset will become a widespread need as more applications in the telecommunications business will be programmed using CORBA. Given the large investment of carriers in CMIP/SNMP, however, there will still be a need to manage CMIP/SNMP agents

[1] IBM Research Division, Zurich Research Laboratory, Säumerstrasse 4, 8803 Rüschlikon, Switzerland. Email: {lde, bba}@zurich.ibm.com, WWW: http://www.zurich.ibm.com/~{lde, bba}/.
[2] Universität Bern, Institut für Informatik und angewandte Mathematik, Software Composition Group, Neubrückstrasse 10, 3012 Bern, Switzerland. Email: deri@iam.unibe.ch, WWW: http://iamwww.unibe.ch/~deri/

in the future. If CORBA can be employed to transparently manage those, then a smooth transition/cooperation between the two worlds can be achieved. If a strategy of a company is to move towards CORBA, then they can still access their legacy agents, maybe gradually phasing them out and replacing them by CORBA applications.

There are several, partly orthogonal, partly overlapping, approaches that use CORBA for CMIP/SNMP management which will be outlined and compared in this paper. We present two schemes that use CORBA in a dynamic manner for network management and contrast them to static approaches. The structure of the paper is as follows: first, we will give a short overview of CORBA. Then, a static and two dynamic approaches to use CORBA for network management will be presented and compared with each other, with the focus being on the dynamic approaches. Finally, some conclusion will be drawn.

2. CORBA Overview

CORBA allows instances to be created locally or remotely either in local host or on a remote one. If the instance is remote it can be accessed from every CORBA compliant client, whereas local instances can be accessed only by the application that created it.

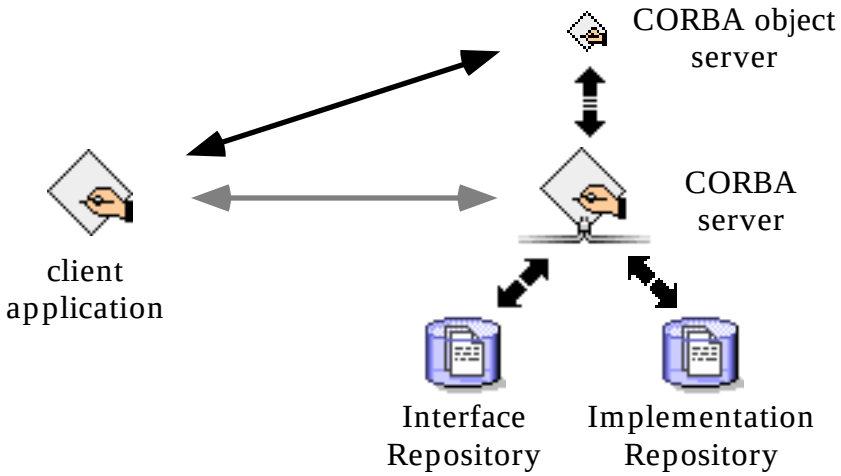

Figure A: Components of CORBA runtime application environment

On each host on which CORBA instances are to be created there is a daemon called CORBA server which is responsible for handling object creation/deletion and other requests sent by client applications[4]. In order to serve such requests, the server accesses two databases: interface repository and implementation repository. The interface repository is a persistent type repository of objects representing the elements of interface definitions and is created and maintained based on information supplied in the IDL source file.

The implementation repository is a database which contains the implementation definitions of CORBA objects, i.e. the shared information about the location of the libraries which implement the CORBA objects and the classes which are offered by a server. Whenever a client application creates a remote object, the server uses the implementation repository in order to have access to the code that implements the requested CORBA object and starts a server application where the CORBA object gets instantiated. The client application receives back from the object creation a proxy pointer, allocated in the client's address space, which points to the real instance created in the server application. Due to this, clients are not allowed to manipulate instances directly but they do access them transparently through the proxy instances. Instead, if the instances are created locally, i.e. in the client address space, there is no interaction with the server and the instances behave like a normal non- CORBA instance. Nevertheless if the instance is created locally, the instance is "private" to the process hence there is no way for external processes to access it.

CORBA interfaces are processed as follows:

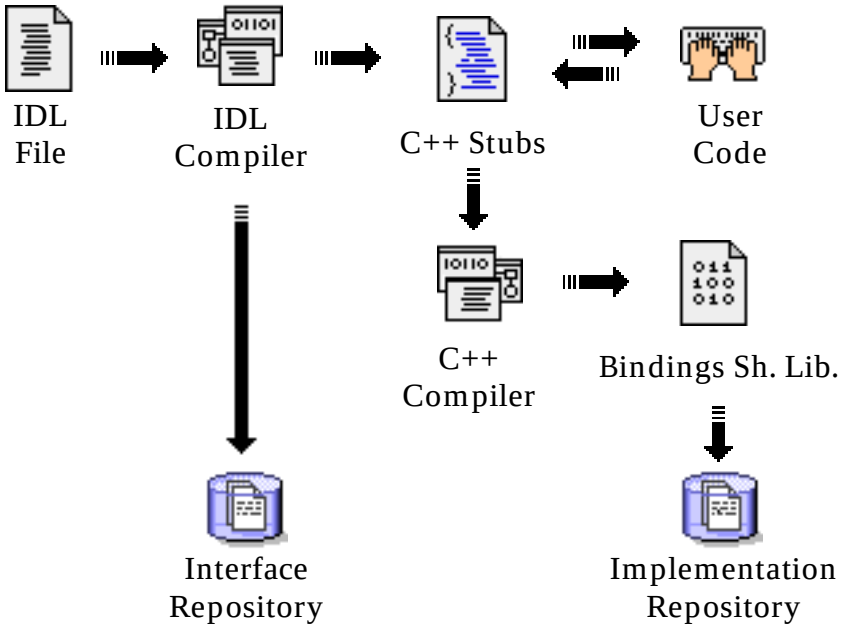

Figure B: CORBA Interfaces Implementation using the C++ language

The IDL compiler compiles the IDL file, updates the interface repository and generates C++[5] stubs for each interface method. The stubs are empty hence they need to be implemented by the user. Once this step has been done, the stubs are compiled and a shared library is generated. Such library is then added to the implementation repository. It is now possible create CORBA instances.

[5] In our case the C++ emitter has been selected. Similar considerations can be done for other languages such as C, Java or

3. Using CORBA for Network Management

3.1. Static Approach

In this section we will present a static mapping method which is currently being defined by X/Open.

3.1.1. XoJIDM

X/Open's Joint Inter-Domain Management task force (XoJIDM) is working on the mapping between GDMO/ASN.1 and IDL and vice-versa (only the first mapping is of interest to us here). Their approach consists of two parts: the first is Specification Translation [Spec95] and defines the static translation of GDMO/ASN.1 to IDL while the second is called Interaction Translation [Int95] and deals with how the mapping is used at runtime. The goal is to translate the MIB of an agent (GDMO/ASN.1) to CORBA IDL which can subsequently be used to manage the agent using CORBA. The approach is shown in the figure below:

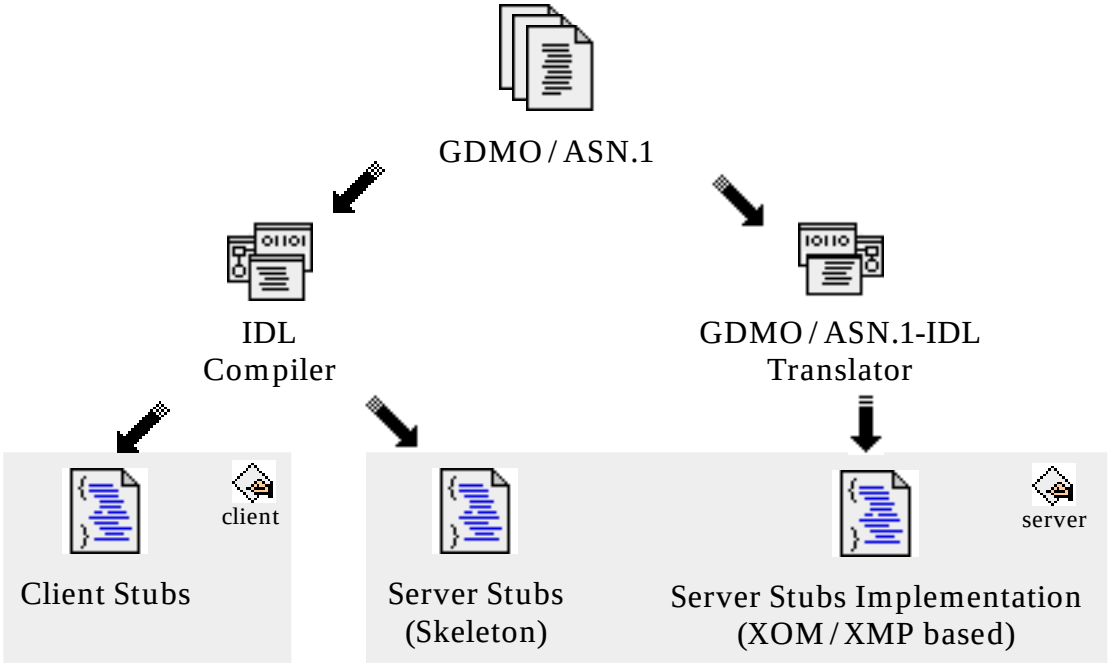

Figure C: XoJIDM Specification and Interaction Translation

The GDMO/ASN.1 documents describing the agent's MIB are translated to IDL and then to a server implementation. GDMO templates are mapped to IDL interfaces and actions and attributes to operations where for each GDMO attribute, a potential GET-, REPLACE-, ADD-, REMOVE-, and SET-TO-DEFAULT-operation is defined. Each ASN.1 type is translated to a corresponding IDL type, e.g. SEQUENCE is mapped to an IDL struct, OCTET STRING to string etc.

IDL attributes generated from ASN.1 types. IDL is then compiled to produce the client and server stubs in the desired language binding (e.g. C++). The server stub and the implementation code generated by the GDMO/ASN.1-IDL compiler are compiled and linked to produce a CORBA implementation. The client stubs are compiled and linked with the user application. Information about available CORBA interfaces (which represent CMIP instances) is contained in the generated client stubs and therefore known to the client at compile time. At runtime, the client proxies will forward any request they receive to their corresponding objects in the CORBA server. These will use the implementation code generated by the GDMO/ASN.1-IDL compiler to communicate with the managed objects in an agent using CMIP/SNMP (e.g. by using the XOM/XMP API [XOM][XMP]).

3.2. Dynamic Approaches

The limitations of the static CMIP/SNMP to CORBA mappings and their extreme complexity has been the main reason that pushed the authors towards a more dynamic approach to the problem. In this section we will present two dynamic models defined by the authors which use CORBA for network management.

3.2.1. GOM

The Generic Object Model (GOM) [Ban] is a framework for management of instances of multiple object models such as CORBA, CMIP or SNMP.

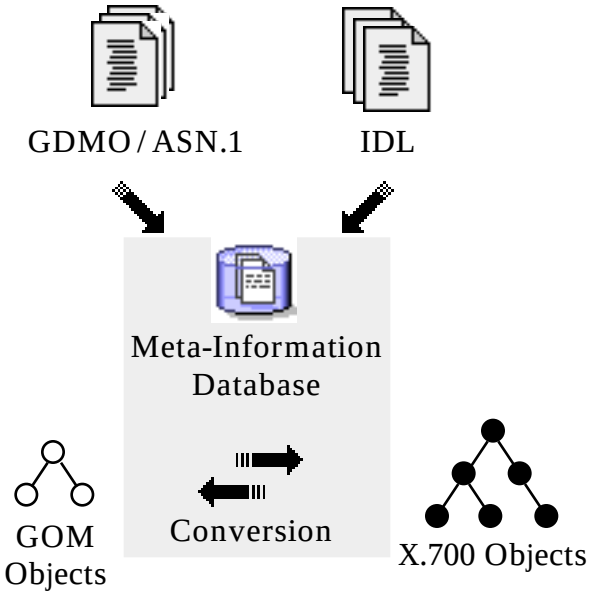

Figure D: GOM Architecture

It uses the concept of reification, modelling elements of object-oriented models as objects themselves [Maes]. Thus, all CORBA interfaces or GDMO

the generic GOM class `GenObj`, all attributes to instances of `Attribute` and all values to instances of `Val`. The interface of class `GenObj` is shown below:

```
class ArgList;
class Val;
class Attribute; // contains Val

class GenObj {
   private:
    List<Attribute> attributes;
   public:
    Val*  Get(const char* attrname);
    bool  Set(const char* attrname, Val& new_val);
    Val*  Execute(const char* opname, ArgList& args);
    [...]
};
```

It has a list of attributes and operations which are instances of the classes `Attribute` and `Operation` respectively. Since client applications do not have to include type information about available classes at compile-time, they typically have a very small size. Also, when a specification is modified, clients do not have to be recompiled, but it is only the modified specification that has to be parsed and fed into the Metadata Information Database (MID) (c.f. below). Once it is there, adapters (see below) that use that specific type information just have to flush their metadata cache in order to access the modified type information, which is a runtime operation. Having no tight binding between client and server and providing a reified object model has a number of advantages especially in X.700 which is more complex than CORBA. It is for example possible to accommodate conditional packages on a as-needed basis, which means that no elements of conditional packages will be initially available when an instance is created. When a request is sent to the instance referring to an attribute residing in a conditional package, the attribute will be created ad hoc using the MID and will be added to the instance's attribute list. This scheme of on-demand loading allows for memory-sensitive applications to be created. Another example is the use of the X.700 ANY DEFINED BY type which is a type that can be determined only at runtime using metadata. It is simply not possible to map this type at compile time to an IDL interface or C++ class in the static approach. As shown above, there is one proxy instance of `GenObj` in the client's address space for each underlying object, be it a CMIP, SNMP or CORBA object. Requests sent to those proxy instances will be forwarded to an adapter which knows how to translate between the generic and exactly one specific model.

Since, contrary to the static approaches described above, there is no compiled-in knowledge of any classes in the system, adapters rely entirely on metadata about the CORBA, SNMP or CMIP classes available to perform their work. GOM includes a MID which is a repository of type information about the various specific models. It is populated by compilers, e.g. in the case of

CORBA, an IDL compiler will parse the IDL specifications and add metadata about CORBA interfaces, their operations and attributes to the MID. In the case of X.700, a GDMO compiler will add metadata about GDMO templates, their operations and attributes and an ASN.1 compiler will provide information about the ASN.1 types. Conceptually, the MID is a single logical database, whereas it uses a separate database for each object model internally. The MID can also be used for other purposes such as lookup of type information for interpreters or debuggers, documentation of class specifications etc.

3.2.1.1. GOMscript

In order to handle instances of GOM, a C++-like interpreter has been written which lets users create, access and delete instances either interactively or by running scripts. GOMscript[6] has simple values such as numbers, booleans, strings and aggregate values such as structs, unions and lists. It has the usual control statements for repetition (`for`, `while`) and conditional branching (`if`, `else`) and is object-oriented in the sense that it implements (single) inheritance, polymorphism and encapsulation. Its core is very small and can be extended (additional functions and classes) through user-written components [Deri95] which are located in shared libraries. GOMscript can be started in daemon mode in which it waits for (potentially remote) clients to send script to be executed. This allows for implementation of simple roaming agents [Mage96] which are essentially scripts moving from machine to machine and taking their state with them. The architecture is shown below:

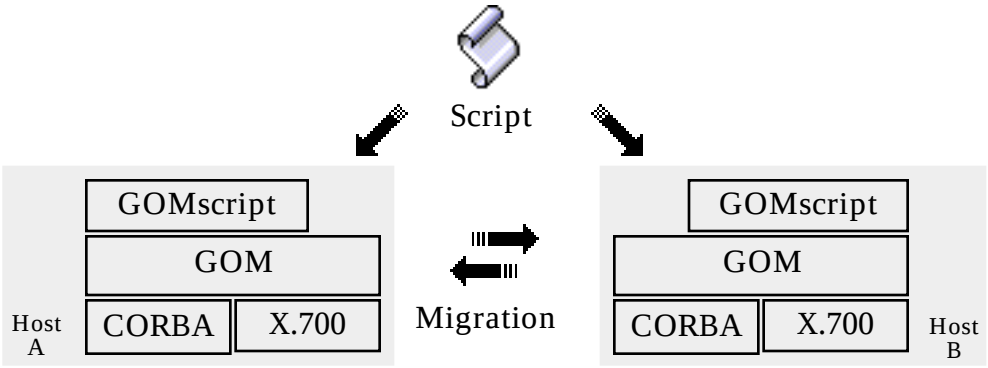

Figure E: Fig. 8: GOMscript as an Execution Platform for Roaming Agents

At any time during its execution, a script may decide to migrate to a different machine. To do so it simply calls a GOMscript function which has as parameters the name of the target host and the port number. This will pack the currently running script and the state (a dump of the symbol table) into a TCP message which is subsequently sent to the target machine. There, another interpreter receives the message, sets its initial state from the dumped symbol table data and starts execution of the script in a separate process.

Since CORBA object references are values in the symbol table that can be dumped to a data stream as well, it is possible for roaming agents to have access to some `global` CORBA instance to which they can refer during their trips[7]. An example of a script for a roaming agent is presented in the next section.

3.2.2. CORBA-Liaison

Liaison [Deri96a] is a software application which allows to manage CMIP/SNMP resources through the Web. The core element of Liaison is the Proxy server which is based on a special type of software components called droplets [Deri95] that have the ability to be replaced and added at runtime allowing to dynamically modify and extend the behaviour of the application that contains them. Among the droplet part of Liaison's standard configuration, there are some which allow Java/C++ applications to do network management through Liaison. Basically Liaison provides some Java/C++ classes, called external bindings [Deri96b], which are linked to the C++ application or Java applet and which allows management operations to be performed. The management application uses the external bindings as normal classes, invoking methods, creating/deleting instances. Transparently, external bindings communicate with the Proxy server using the HTTP [HTTP] protocol, used extensively in the Internet by the World-Wide Web.

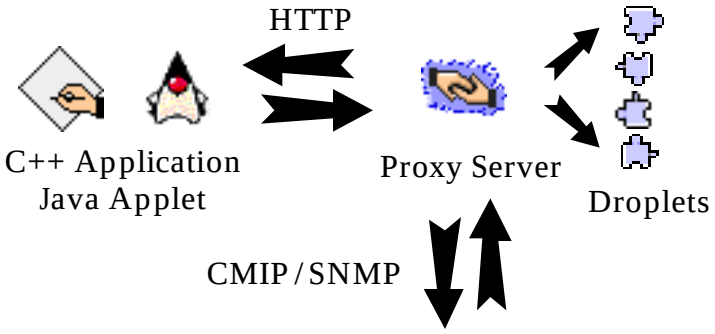

Figure F: Liaison's Java/C++ Bindings

Whenever a method of the external bindings that performs a management operation (e.g. CMIP M-SET) is called, an HTTP request is sent to the Proxy. Such an HTTP request contains all the parameters necessary to issue the management request. The Proxy issues the management request, receives the response(s) and handles all the possible errors. Once the operation has been completed, an HTTP response containing the result of the operation is sent back to the application. This mechanism allows external applications to do network management in a simple and effective way by exploiting Liaison and without having to handle the complexity of management protocols. In fact, external bindings deal only with object oriented concepts shielding

completely the underlying management protocol.

Based on external bindings some CORBA interfaces to the CMIP/SNMP protocols have been defined. An important design choice has been to do not map each CMIP/SNMP object to a CORBA object like seen in other translation methods, but to map the CMIP/SNMP protocols to CORBA in a very generic way. This choice has been motivated by the following reasons:

- ability to fully support CMIP/SNMP from CORBA
- low complexity and flexibility since there is no need to generate new CORBA classes for new CMIP/SNMP objects that need to be supported
- user-defined abstraction level: users can define further CORBA classes based on the basic ones depending on their needs without having to pay the complexity of having a CORBA class for each CMIP/SNMP object even if not all of them are currently used.

CORBA-Liaison interfaces (CL) for CMIP/SNMP, defined using the IDL (Interface Definition Language) language, have been implemented using DSOM [DSOM], IBM's CORBA compliant ORB (Object Request Broker). Since we do not rely on any specific characteristic of DSOM, similar considerations can be done for other CORBA implementations. ASN.1 datatypes, like external bindings datatypes, have been mapped to strings, hence to the native `string` IDL datatype. CL interfaces representing CMIP and SNMP objects, defined in a way very similar to Liaison's external binding, are depicted below:

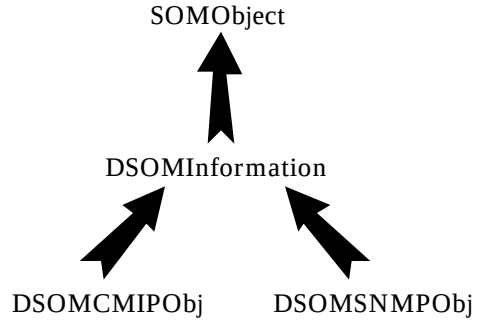

Figure G: CL Interfaces for CMIP/SNMP

The interface `DSOMInformation` contains the information relative to the request and to the response(s). Internally the values are stores in a hash table where the attributeId constitutes the key and the attributeValue the value of each table entry. The use of hash table associated with the mapping of values into strings allows to handle objects independently from their class and complexity. In the case of CMIP, the presentation layer or a thin layer on top of the stack, converts attribute values into strings and vice–versa, whereas in the case of SNMP is the Proxy that handles the conversion.

```
interface DSOMInformation: SOMObject {
    void     SetAttribute(in string name, in string value); // Stores a value into the table
    string   GetAttribute(in string name);            // Retrieves a value
    void     RemoveAttribute(in string name);         // Remove an attribute from the table
    sequence <string> GetAttributes();                // Returns all the attribute values
    sequence <string> GetAttributeKeys();             // Returns all the attribute keys
    void     RemoveAllAttributes();                   // Table clean up
    [...]
}
```

Being `DSOMInformation` built upon a hashtable, it is possible to retrieve and store elements efficiently and to have only a few methods that handle all the situations. In order to further simplify the attribute manipulation, the classes `DSOMSNMPObj` and `DSOMCMIPObj` have been defined. These classes simplify the access to `DSOMInformation` by defining some sort of macros such as `DSOMCMIPObj::GetObjectInstance()` which are mapped into calls to `DSOMInformation` methods (`DSOMInformation::GetAttribute("objectInstance")` in this case). C++ methods of CL are almost identical to the ones defined in the corresponding class part of the external bindings hence the stub implementation has been very straightforward. The similarity between these interfaces and the corresponding classes part of the external bindings has the advantage that developers can use both DSOM and the external bindings, having to learn just one object model. Additionally, code can be written once and them slightly modified to use either the C++/Java external bindings or the C++/Java language bindings of the DSOM interfaces. This is because methods and classes have the same names and parameters. The following example shows how a simple program based on the external bindings can make use of the DSOM interfaces by simply adding the code shown in bold. The code fragment below reads the value of the attribute `systemTitle` of the instance `genericNetworkId=Net1@systemId=(name Telco)` using the CMIP protocol.

```
Environment ev;
SOM_InitEnvironment(&ev); SOMD_Init(&ev); /* Initialization */

try {
 Liaison_DSOMCMIPObj *cmip = new Liaison_DSOMCMIPObj(&ev);
 cmip->UseProxy(&ev, "adl.zurich.ibm.com"); /* Uses the Proxy server running on host adl */
 cmip->SetAgentAET(&ev, p, "MIBCTL" /* CMIP Agent AE-Title */);
 cmip->SetObjectClass(&ev, "system");
 cmip->SetObjectInstance(&ev, "genericNetworkId=Net1@systemId=(name Telco)");
 cmip->SetAttribute(&ev, "systemTitle", "");
 cmip->CMIPGetAttributes(&ev); // Issue the CMIP M-GET request

 if(somExceptionId(&ev) == NO_EXCEPTION)
    printf("systemTitle is: %s\n", cmip->GetAttribute(&ev, "systemTitle"));

 delete cmip;
} catch(char *exc) { printf("Caught exception: %s", exc); }

SOMD_Uninit(&ev); SOM_UninitEnvironment(&ev); /* Termination */
```

Basically the only code that has to be added is related to:

- the DSOM initialization/termination

- the `Environment` parameter needed in each DSOM method call
- exception handling that cannot catch all the DSOM exceptions using the try/catch mechanism since DSOM may use the `Environment` parameter to report about error conditions.

The design choice to implement the DSOM stubs using the external bindings instead of wrapping the whole Proxy into a DSOM object has the following advantages:

- since the external bindings are quite light, the DSOM interfaces implementation is very light (less than 70Kb)
- DSOM has to be installed only by users that need to access the Proxy using DSOM, i.e. applications based on external bindings do not need to have DSOM installed in order to run
- DSOM allows to create objects on hosts where the Proxy is not installed exploiting a remote Proxy, without the need to have DSOM installed on the host where such Proxy runs (the communication DSOM server/Proxy is HTTP based)
- depending on the situation, users can decide to access services provided by the Proxy using HTTP, DSOM or both (if the Proxy would have be wrapped in a CORBA object then users would need DSOM to access the Proxy)
- it is possible to manage hosts outside firewalls using a local Proxy and DSOM interfaces since they are based on HTTP (DSOM cannot cross firewalls, HTTP can).

The drawback of this solution is that every time a management operation need to be issued, there is a communication between the DSOM client, the DSOM server and the Proxy instead of having the Proxy contained inside the DSOM server. In the tests we have performed, the slowdown of the proposed solution is not more than 10-20% with respect to a full integration of the Proxy inside a DSOM object. Considering the many advantages of this solution with respect to a total DSOM integration, we believe that this overhead is acceptable and almost neglectable if client applications can perform multiple operations concurrently (multithread) without active wait.

3.2.2.1. Managing CORBA-Liaison Interfaces with GOMscript

Using GOMscript, CL interfaces can be handled interactively or by running scripts, thus allowing for simple management tasks to be automated. The code using the C++ language bindings of Liaisons CORBA interfaces shown previously, can thus be rewritten as follows (using script form):

```
system_name="genericNetwokId=Net1@systemId=(name Telco)";
```

```
  cmip.UseProxy("adl.zurich.ibm.com");
  cmip.SetObjectClass("system");
  cmip.SetObjectInstance(system_name);
  cmip.SetAttribute("systemTitle", "");
  cmip.CMIPGetAttributes();

  if(GetEx() == NULL) {
      println "systemTitle is " + cmip.GetAttribute("systemTitle");
  }
```

GOMscript can make use of the Liaison CORBA interfaces for implementing simple roaming agents for network management tasks. Agents may for example be sent to different locations to perform data collection and filtering tasks the result of which are periodically sent back to a central CORBA instance. A sample script is shown below:

```
if(hosts == NULL)
  hosts=#("adl", "saz", "mut", "kis");  // list of hosts to visit

if(collector == NULL) {
  collector=new Obj("CORBA", "Collector", "adl"); // "adl" is central loc.
}

if(hosts.Length() > 0) {
  target_host=hosts.At(0);
  hosts.Delete(0); // otherwise we loop since the script is always
                   // sent to the same location !
  SendAgent(target_host, 10000, "", ""); // will know 'hosts' and
                                         // 'collector' at target location
}

/* Now start the assigned task */
snmp_obj=new Obj("", "::Liaison::DSOMSNMPObj", ""); // create local snmp obj
snmp_obj.SetSnmpAgentAddress("", 160);              // local snmpd is used
hostname=GetHostname();
while(true) {
  out_requests=snmp_obj.GetAttribute("ifOutRequests.0");
  // get more interesting data ...
  collector.UpdateAttr(hostname,          // primary key
                       "ifOutRequests.0", // secondary key
                       out_requests);     // data
  Sleep(60); // sleep 1 minute
}
```

First the script is sent to a set of machines determined in `hosts`. As long as the list is not empty, the script will propagate a copy of itself (and its current state) to the next member of the list. Having done this, the assigned work can be started. An SNMP object is created and a value retrieved (`ifOutRequests.0`). This value is then sent to a central CORBA instance where it may be accessed by other clients for examination. Since values can be summarised, correlated or filtered by an agent before sending them to the global CORBA instance, a sort of event filtering mechanism could be easily implemented.

Since different sets of classes and functions may be available on various machines in the network, GOMscript allows to check for the existence of classes and functions before usage. Also, since GOMscript is based on GOM, it is possible to dynamically find out what classes are available on a certain system and use their services. Consider the case of a `Printer` class which offers a service named `Print`: an agent can always invoke this service on any object that offers a service of the same signature regardless of whether the class is derived from some common base class or the agent has (compiled-in) knowledge of the `Printer` class.

4. Static vs. Dynamic Mapping

The work done on using CORBA for network management as described above can be broadly divided into two categories: approaches which statically translate GDMO/ASN.1 to generate code which is included by management applications that therefore know at compile time the extent of classes that they can handle, and dynamic approaches without dependency on compile time knowledge since they are either string- or metadata-based. The following table lists some of the major differences between the various approaches:

	CORBA-Liaison	GOM [Ban]	XoJIDM [Int95][Spec95]
Mapping Type	Dynamic	Semi-dynamic	Static
Typing	Untyped	Runtime-type checked	Strong
Type Checking	Runtime (by Proxy and the OSI Stack)	Runtime (using metadata)	Compile Time
Implementation size	Small (<70Bk irregardless of the type/number of managed objects)	Medium (it needs a metadata repository)	Large (it includes a mass of generated types/methods)
ASN.1/CORBA type mapping	All datatypes are mapped to a string	Datatypes are mapped to a small set of GOM types (15)	Every datatype is mapped to an equivalent CORBA type (sometimes more than one)
CMIP/SNMP:CorbaClasses	N:1	N:15	N:M (N <= M)
CMIP Support	Yes	Yes (except for M-EVENT-REPORT and M-ACTION)	Yes
SNMP Support	Yes	No (no SNMP adapters currently available)	Yes

The CORBA Liaison (CL) interface approach is completely untyped since all types are mapped to strings. Conversion between strings and the desired data

type of the host language (e.g. C++) has to be done by the programmer. This may be easy for simple types such as strings or numbers, but increases complexity for the programmer considerably for aggregate types such as structs or sequences. Also, probability for introduction of errors in user-written conversion functions increases. This trade-off, however, was accepted by CL because its main goal was the creation of a light-weight model for network management that should be flexible (no compiled-in knowledge) and that should be integrated with the World Wide Web [HTTP] which uses strings as the major data type anyway. Also, network management is nowadays still predominantly based on SNMP which uses mainly atomic data types such as strings or integers. The programmer specifying the types in a string-based syntax which will be runtime checked by the Proxy in the case of SNMP, or by the OSI stack in the case of CMIP. Compared to the static approaches with their strong typing enforced at compile time, GOM enforces typing at runtime using metadata. Contrary to CL, which knows only the string type, it has types for representation of classes (`GenObj`), attributes (`Attribute`) and values (`Val`, `Integer`, `String`, `Struct`, `Sequence` etc.). Whereas CL maps all types to strings, GOM maps them to an instance of this set of fixed types, and the static approach maps each type to a corresponding IDL type.

Whereas the static approaches fully integrate the translated code into the target type system using the target's native types (e.g. C++), GOM offers an abstraction of the target's type system (ca. 15 types) as API to the user whereas the API of CL is the single type `string`. In the case of the static approaches, the client of the API may mix types of the target system and the generated code since they are the same whereas using GOM, native (C++) types have to be converted to/from GOM types (e.g. `int` to instance of `Integer`)[8]. Using CL, it is the responsibility of the client to convert the `string` types to/from the native type system (C++, Java etc.). The dynamic approaches have two major advantages over the static ones: they typically produce smaller client applications and they are much more flexible. Since clients do not possess a-priori knowledge about classes available in the system, but rather use strings or metadata, they are independent from class modifications and can continue working while clients using the static approach may need to be recompiled. This is an essential asset in areas such as topology browsers and roaming agents (c.f. above) that do not know all the classes they will encounter when compiled. Including at compile time a fixed set of classes may yield potentially large client applications that have to pay (in terms of size) for all the classes they carry with them even if only a few are actually used. In the dynamic approaches, when a client needs to handle a new/modified class, either the latter's metadata is dynamically loaded (GOM) or it needn't be loaded since a string type represents all types (CL). An area where metadata is useful or needed are X.700's conditional packages, attributes and the

ANY/ANY DEFINED BY ASN.1 types which can only be resolved to a correct type at runtime.

5. Conclusion

This article shows that management of OSI and SNMP network resources through CORBA is possible and feasible. The two main directions of research currently done on using CORBA with network management have been analysed and compared. Besides known techniques, this paper described a novel technique named CORBA-Liaison. Relevant characteristics of this technique are: efficient, full CMIP/SNMP protocol support, light and simple object model independent from a particular CORBA implementation.

The goal of this paper was not to demonstrate that one approach is better than another but to understand the benefits and limitations and then to identify the solution that better fits user requirements. Finally this paper has shown that CORBA-based SNMP/CMIP management is now becoming a mature technique which overcomes most of the limitations of early solutions.

6. References

[ASN1] ISO/IEC, CCITT, Specification of Abstract Syntax Notation One (ASN.1), ISO/IEC 8824, CCITT Recommendation X.208, 1988.

[Ban] B. Ban, Towards a Generic Object-Oriented Model for Multi-Domain Management, Proceedings of ECOOP '96 Workshop on Systems and Network Management, Linz, Austria, July 1996

[CMIP] ISO/IEC, CCITT, Information Technology-OSI, Common Management Information Protocol (CMIP)-Part 1: Specification ISO/IEC 9596-1, CCITT Recommendation X.711, 1991.

[Deri95] L. Deri, Droplets: Breaking Monolithic Applications Apart, IBM Research Report RZ 2799, September 1995.

[Deri96a] L. Deri, Surfin' Network Management Resources Across the Web, Proceedings of 2nd Intl. IEEE Workshop on Systems and Network Management, Toronto, June 1996.

[Deri96b] L. Deri, Network Management for the 90s, Proceedings of ECOOP '96 Workshop on Systems and Network Management, Linz, Austria, July 1996.

[DSOM] IBM Corporation, DSOM Development Toolkit, October 1994.

[Hierro] J. Hierro, Architectural Issues For Using CORBA Technology in OSI Systems Management, Append of draft to XoJIDM forum, August 1994

[HTTP] T. Berners-Lee, R. Fielding and H. Nielsen, HyperText Transfer Protocol-HTTP/1.0, Internet Draft, 10/16/1995.

[Int95] Joint Inter-Domain Working Group, X/Open and Network Management Forum, Inter-Domain Management Specifications: Preliminary CORBA/CMISE Interaction Translation Architecture, April 1995.

[Maes] P. Maes, Concepts and Experiments in Computational Reflection, Proceedings of the 2nd OOPSLA Conference, 1987, pp. 147-155.

[Mage96] T. Magedanz and T. Eckardt, Mobile Software Agents: A New Paradigm for Telecommunications Management, Proceedings of 2nd Intl. IEEE Workshop on Systems Management. Toronto, Ontario, 1996.

[OMG] Object Management Group, The Common Object Request Broker: Architecture and Specification, Revision 2.0, July 1995.

[SNMP] J. Case, M. Fedor, M. Schoffstall and C. Davin, The Simple Network Management Protocol (SNMP), RFC 1157, May 1990.

[Spec95] Joint Inter-Domain Working Group, X/Open and Network Management Forum, Inter-Domain Management Specifications: Specification Translation, April 1995.

[XMP] X/Open Company Ltd., Systems Management: Management Protocol API (XMP), X/Open Document C306, March 1994.

[XOM] X/Open Company Ltd., OSI-Abstract-Data Manipulation API (XOM), X/Open Document C315, February 1994.