Tcl-MCMIS: Interpreted Management Access Facilities

Thurain Tin, George Pavlou, Rong Shi

Department of Computer Science
University College London
Gower Street, London, WC1E 6BT, UK

e-mail: thurain@cs.ucl.ac.uk

Abstract. Programming OSI management communications is considered daunting because of the complexity of the service/protocol (CMIS/P) and the lack of standardized high-level Application Program Interfaces (APIs) that can harness the power and hide the protocol and abstract syntax complexity. In the OSIMIS platform, high-level C++ APIs, namely the Remote MIB and Shadow MIB, are provided to support the construction of manager applications using object-oriented concepts. Despite this level of support, there are times when the application programmer needs to undertake rapid development away from the compiled approach, using only a simple scripting language. In OSIMIS, Tcl-MCMIS is one such scripting language extension to Tcl, realising an interpreted CMIS manager interface, which can easily be intermixed with the Tk widgets to construct management graphical user interfaces. This paper describes how Tcl-MCMIS is designed in a generic way using the existing APIs to retain the full power of the underlying protocol support.

Keywords: OSI, Platform, CMIS/P, API, Scripting Language

1. Background and Introduction

The OSI management technology [X701] provides a powerful framework based on the rich systems management functions (SMFs) supported by the object-oriented specification methodology (GDMO) [X722], the expressiveness of the associated management information structure (MIM, DMI) [X720] [X721] and the complexity of the communication service and protocol (CMIS/P) [X710] [X711]. Following a fully fledged object-oriented approach, the framework lends itself naturally to object-oriented design and realisation, and the OSIMIS platform [Pav93] [Pav95] is a management information service infrastructure enabling this technology with the provision of well-defined C++ [Strou] Application Program Interfaces (APIs) and supporting tools. The APIs support both managed object realisation (agent role) and high-level distributed access capabilities (manager role), relieving the application programmer from handling directly the "raw" protocol access and plain ASN.1 syntax manipulation (encoding/decoding). Thus there is a clear separation between the generic and specific parts of the management applications as shown in Fig. 1.

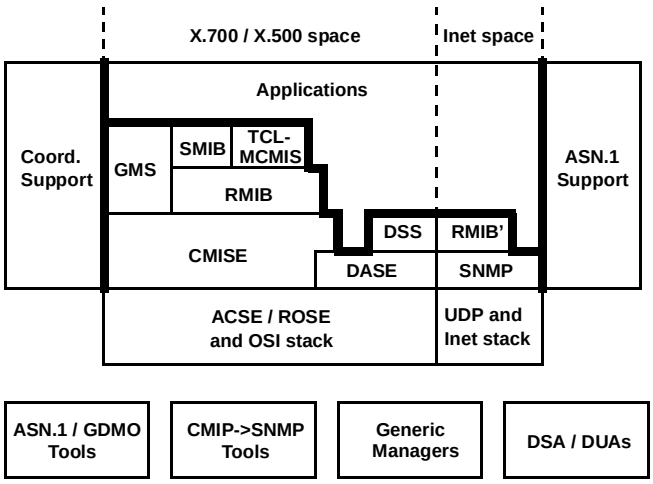

Fig. 1. OSIMIS Layered Architecture, Tools and Generic Applications

In OSIMIS, the construction of high-level distributed access for applications in manager roles is primarily achieved with the Remote MIB (RMIB) API [Pav94]. RMIB provides an object-oriented abstraction of the remote OSI MIB by proxying the agent concerned in the managing application. Management operation requests to the agent are carried out by invoking the appropriate C++ method calls of the representing proxy object in a string-based notation. The proxy object is created as an instance of the class *RMIBAgent* on a per-association basis and this class hides all of the low-level CMIS/P access details and provides the transparent Protocol Data Unit (PDU) assembly for the linked replies. Asynchronous flow of information is possible using the *RMIBManager* abstract class with a number of call-back mechanisms. The latter class is abstract (virtual) to allow specialisation to take place in a derived manager class with the required behaviour that processes the event-driven and asynchronous results.

While the RMIB harnesses and hides the protocol complexity through encapsulation and a string-based interface, the Shadow MIB (SMIB) API [Pav94] extends the model of RMIB and addresses the dynamics of the management information flow and the containment tree (MIT) traversal. SMIB uses the abstraction of objects in local address space, "shadowing" the managed objects of the remote MIB. The advantages are two-fold: first, the API can be less CMIS-like and more intuitive in terms of pointers in local address space, and second, the existence of managed object images as shadow managed objects (SMOs) can be used to cache the information and therefore minimize the communication with the agent. The RMIB and SMIB models are depicted in Fig. 2.

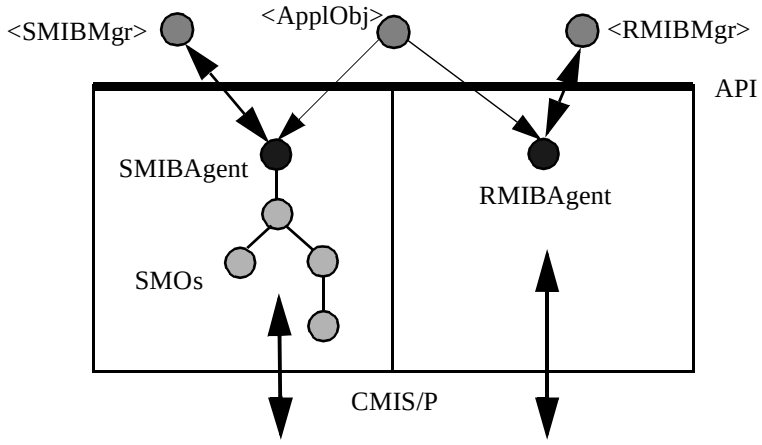

Fig. 2. The Remote and Shadow MIB Access Models

High-level abstractions are also used to provide the support for dealing with ASN.1 in a way that completely shields the application programmer from the actual abstract syntax. This is achieved by wrapping up the internal C data type representations in C++ classes using an ASN.1 meta compiler. Essentially this employs polymorphism and encapsulation of behaviour in data types with respect to encoding and decoding which take place in a transparent fashion.

The RMIB and SMIB APIs are compiled and as such they exist as libraries. Although these APIs greatly reduce the amount of work needed by the application programmer by factorising out the generic parts of the management application, their compiled nature, however, does not facilitate rapid prototyping. Therefore there is a clear need for an interpreted interface and this requirement seems most imperative as prototyping and scripting language environments commonly exist with the support for building graphical user interfaces (GUIs). The Tool Command Language (Tcl) and the associated Tk widget set from the University of California at Berkeley have the basis for the quick construction of simple test scripts with X-Windows support [TclTk]. The core of Tcl has been extended, by a large user community, to provide a number of useful "extensions" typically through C and C++ linkage. In OSIMIS, we have researched, designed and implemented a CMIS-based Tcl extension set that is suitable for prototyping OSI manager applications, called Tcl-MCMIS. In this paper, we explain how the design of a simple scripting language interface like Tcl-MCMIS can be implemented in a generic fashion and also discuss how the interface can be made to operate across a number of different transport mechanisms, independent of the OSI stack.

2. Rationale and Reusability

The advent of the Tcl scripting language and of the associated Tk widget set has greatly simplified the construction of test scripts and GUIs, both being of paramount importance to network and system management environments. In OSIMIS, every management syntax has a well-defined string representation used mostly for pretty printing but also potentially for programming. The syntax support is hidden behind an O-O ASN.1 API, through the *Attr* (general ASN.1 type) and *AVA* (Attribute Value Assertion - type/value pair) classes, where every attribute, action, error and event report value is encapsulated in a C++ class. This together with the string-based CMIS interface from RMIB led to an interpreted interface in Tcl, since the latter naturally uses the string representation as its data type.

Also quite importantly, OSIMIS already has a set of generic manager applications which are built with no compiled knowledge of the agent's GDMO. These generic applications operate from the Unix command line / shell and as such the given string arguments are parsed from the environment to the program code. It is thus natural enough to reuse part of the generic manager applications to establish a similar command line syntax in Tcl-MCMIS. Unlike the compiled approach, application programmers with little or no exposure to network programming should be able to perform powerful management operations with just a few simple lines of scripting commands, and redirect the results to a number of display widgets in Tk.

3. Design and Implementation

Keep it simple: as simple as possible, but no simpler. - Albert Einstein

In designing the Tcl-MCMIS interface, we took the following steps and targets to help us introduce the new extensions and control the component interactions making use of the existing OSIMIS manager infrastructure.

- define the new command name and associated command syntax
- implement the corresponding command procedure in C or C++
- register the command and procedure names with the interpreter
- control the agent and manager objects for management communications via the RMIB
- format the management results
- asynchronous and event-driven support
- timer scheduling
- interactive support as in *wish*(1)

3.1. Defining a new command name and the associated syntax

Defining a new command name should be simple and straight forward. The name should succinctly identify the purpose of the command. For example, **m_connect** is the command to be used to establish management associations to the remote MIB agents. The syntax associated with a new command should identify the required and optional arguments expected by that command. As mentioned previously, we have followed the style of the generic manager applications and adopted a similar command syntax for Tcl-MCMIS where mandatory arguments precede the optional ones and the latter are identified with the flags. In **m_connect**, for instance, we will need to tell which agent object to use (with the unique agent identifier) for the association, the logical application title (the agent name), the host machine name, and the timeout period to be used in all synchronous management operations. Hence, **m_connect** has the syntax shown below.

m_connect *agentId ?*-a *appl? ?*-h *host? ?*-t *timeout?*

In the table below we list the complete management commands and their syntaxes.

CMIS primitive	Management command and syntax
M-INITIALISE	`m_connect` *agentId ?*`-a` *appl? ?*`-h` *host? ?*`-t` *timeout?*
M-TERMINATE	`m_disconnect` *agentId*
M-GET	`m_get` *agentId ?*`-c` *class? ?*`-i` *instance? ?*`-s` *scope ?sync??* *?*`-f` *filter? ?*`-a` *attr ..? ?*`-m` *managerId ?*`-o??`
M-CANCEL-GET	`m_cancel_get` *agentId invokeId*
M-SET	`m_set` *agentId ?*`-c` *class? ?*`-i` *instance? ?*`-s` *scope ?sync??* *?*`-f` *filter? ?*`-w|a|r|d` *attrType?=attrValue?? .. ?*`-m` *managerId?*
M-ACTION	`m_action` *agentId ?*`-c` *class? ?*`-i` *instance? ?*`-s` *scope ?sync??* *?*`-f` *filter? ?*`-a` *actionType?=actionValue?? ?*`-m` *managerId?*
M-CREATE	`m_create` *agentId ?*`-c` *class? ?*`-i` *instance* `|` `-s` *superiorInstance?* *?*`-r` *referenceInstance? ?*`-a` *attrType=attrValue? .. ?*`-m` *managerId?*
M-DELETE	`m_delete` *agentId ?*`-c` *class? ?*`-i` *instance?* *?*`-s` *scope ?sync?? ?*`-f` *filter? ?*`-m` *managerId?*

Table 1. Management Commands and their Syntaxes

3.2. Implementing the command procedures

In Tcl, every command has its corresponding procedure which typically parses the given command line arguments and realises the behaviour of the command's functionality. The command name and the procedure name are normally registered with the interpreter at initialisation. During run-time, whenever a command is invoked, the interpreter selects the correct command procedure through an internal look-up table and finally executes the procedure using the given arguments from the calling environment.

In Tcl-MCMIS, there are simple command procedures realising the code needed to create/delete proxy agent objects and manager objects, through the **rmib_agent**, **rmib_manager** and **rmib_destroy** commands, which are used to support the management communications. The command procedures implementing management operations are much more complex and they are explained in the next section. Additionally, there are procedures for scheduling and cancelling timer wake-ups.

3.3. Enabling management communications via the RMIB API

Typically, a management operation procedure carries out the required functionality by first looking for, using the unique object identifier(s), the appropriate proxy agent object (an RMIBAgent instance) to communicate with the remote agent and may be the manager object (an RMIBManager-derived instance) if the command was called asynchronously. Next, the given command line arguments are parsed and packaged suitably into C and C++ parameters. Then the corresponding C++ method of the RMIBAgent that provides the CMIS operation is executed with the translated arguments.

For successful synchronous operations, the command procedure completes by extracting the returned C++ result into a consistent Tcl list format, and finally returns the list to the calling environment. If the operation failed locally during interpretation, then an error message is returned immediately with a failure indication to the caller. Fig. 3 shows the general interaction in Tcl-MCMIS. In the case of asynchronous operations, the result formatting takes place, of course, in the manager's call-back environment. The asynchronous and event-driven support is explained later, while the formatting of management results that is consistent in all cases of management operations is explained next.

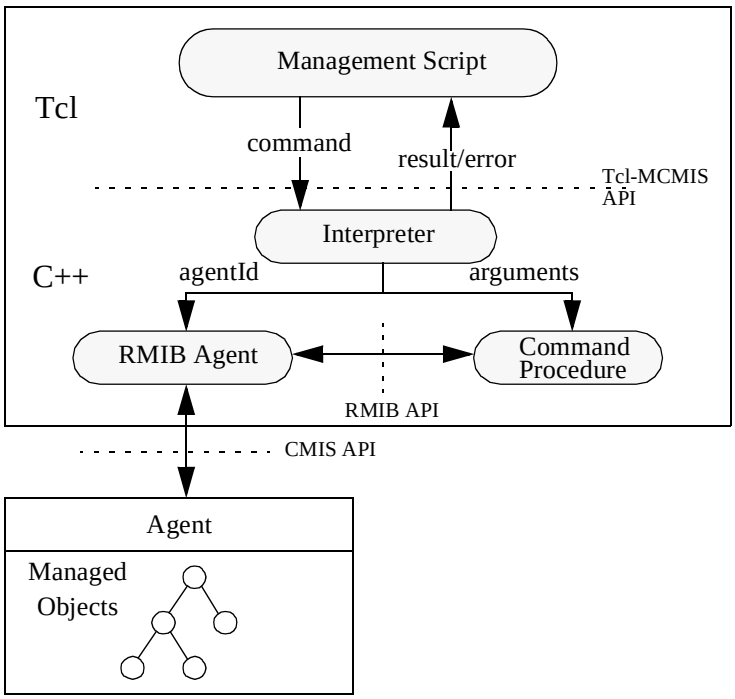

Fig. 3. Tcl-MCMIS Model Interactions

3.4. Result formatting

After invoking one of the management commands, the result of the operation is, whether synchronous or asynchronous, whether or not successful in the remote managed system, conveyed in a consistent list format even though managed object results will vary according to the nature of the operation. This consistency also allows for an easy manipulation of results using Tcl's built-in list commands. The top-level result format and the sub-components are described in the following sections.

3.4.1. {moList}

{moList} is a list which represents the complete result of a management operation. The format is:

$$\{id\ replyId\ appl\ host\ operType\ firstError\ \{mo_1\}\ ...\ \{mo_n\}\}$$

id is either the identifier of the agent object (if synchronous invocation) or the identifier of the manager object (if asynchronous invocation) used in the management command. *replyId* is the reply identifier used in the actual CMIS result and is equivalent to the invocation identifier of the original request. It is also the only means to correlate the asynchronous results. *appl* and *host* identify respectively the logical application title (the agent name) and the host name of the remote managed system in which the operation was performed.

operType is the type of operation performed i.e. one of M_GET, M_SET, M_ACTION, M_CREATE or M_DELETE. *firstError* is the error code of the first managed object in the list with an error (getListError and setListError are excluded as partial results are expected in these cases). Finally, the remaining part of the list consists of one or more {*mo*} lists each of which represents a managed object result.

3.4.2. {mo}

{mo} is a list which represents a single managed object result. The format is:

$$\{errorCode\ \{class\}\ \{instance\}\ \{operTime\}\ [\{ava_1\}\ ...\ \{ava_n\}]\}$$

errorCode is the error code resulting from operating on the managed object. Error codes and their descriptions are described in the appendix. {*class*} is a list containing the class name of the managed object which can be empty {} in the case of certain errors. {*instance*} is a list containing the instance name of the managed object which can also be empty {}. {*operTime*} is the time of operation in the remote managed system and available only when *errorCode* is noError, getListError or setListError, otherwise it will be the empty list {}.

What follows after {*operTime*} depends on the type of operation performed, and if applicable, the kind of error produced. This may be a number of {*ava*} lists representing the attribute value assertions or a single {*ava*} list representing either an action result or error information. In the case of failure, nothing is included beyond {*operTime*} *except* when *errorCode* is noSuchAttribute, missingAttributeValue, invalidAttributeValue or processingFailure. Each of the three variations of {*ava*} is described in the following sections.

3.4.3. {attr}

{attr} is a list which represents an attribute value assertion result in the case of the **m_get**, **m_set** and **m_create** commands. The format is:

$$\{[errorCode]\ type\ [\{value\}]\}$$

Firstly, if the error code in the containing managed object is noError then attr is of the form {*type* {*value*}} where type is the attribute type. {*value*} is the attribute value and is given in a list which may be of a variable length including the white spaces. In the case of set-valued attributes, {*value*} will be of the form {{foo % bar}} using the "%" character to separate the attributes.

Secondly, if the error code in the containing managed object is either getListError or setListError, then attr can be of the form {*errorCode type*} describing an attribute level error. For example, {noSuchAttribute foobar}. In these situations, valid attribute results must be distinguished from those which are not by explicitly checking the first element of {attr}.

3.4.4. {actionRes}

{actionRes} is a list which represents an action result in the case of the **m_action** command. The format is:

$$\{type\ \{value\}\}$$

type is the action type and {*value*} is the action result given in a list which may be of a variable length including the white spaces. For example, {getUserNamesReply { rongshi saleem marvin marvin marvin }}.

3.4.5. {errorInfo}

{errorInfo} is a list which represents the error information *only* when the error code in the containing managed object is one of noSuchAttribute, missingAttributeValue, invalidAttributeValue or processingFailure. The format is:

{*type* [{*value*}]}

The error information is otherwise not conveyed for other kinds of errors because “thinking” manager applications should be able to cope in those situations, and the error code of the managed object should suffice. For example, noSuchObjectInstance error code is self-explanatory with respect to the object instance parameter used in the request.

If the error code is noSuchAttribute or missingAttributeValue, then errorInfo identifies the attribute type in the form {*type*}. On the other hand, if the error code is invalidAttributeValue or processingFailure, then errorInfo identifies the type/value combination of the error information in the form {*type* {*value*}}.

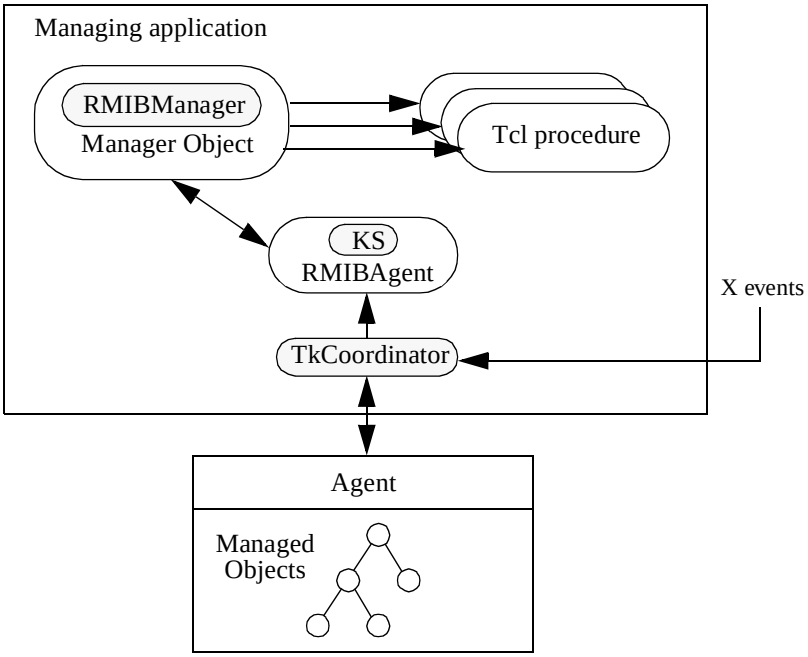

Fig. 4. Event-driven and asynchronous information flow

3.5. Asynchronous and event-driven support

In supporting the construction of any complex management systems, e.g. TMN Operations Systems [M3010], it is very important to provide the asynchronous flow of information and event-driven features like the CMIS event notification service to effectively optimise the communication link, to avoid blocking and receive the notifications of changes from the remote MIB agent. As mentioned, in the RMIB API, the RMIBManager class enables a number of asynchronous and event-driven services via the specialised call-backs.

In Tcl-MCMIS, we provide the **rmib_manager** command to instantiate manager objects of a special RMIBManager-derived class to achieve the “double call-back” as shown in Fig. 4. The first call-back takes place between the RMIBAgent and the RMIBManager, and the second call-back takes place from the RMIBManager to a Tcl procedure. The latter may be a procedure which processes an event report or a CMIS result, or otherwise accept the notification of remote agent’s termination. Typically, the call-back procedures are registered at the creation of a manager instance using the **rmib_manager** command by declaring the procedure names of the call-backs to be used. The application programmer must, of course, ensure that the call-back procedures are properly implemented a priori.

The handling of external input and events is provided in OSIMIS through the Coordinator and Knowledge Source (KS) abstraction, where a derived Coordinator class called TkCoordinator is used in Tcl-MCMIS to allow multiplexing of the incoming data including the X-Windows events of Tk.

3.6. Timer scheduling

In management systems which require polling the managed objects and real resources, it is imperative to provide some form of scheduling timer wake-ups. In OSIMIS, the scheduling mechanism is part of the Coordinator and Knowledge Source abstraction and as such we have made this facility to be part of Tcl-MCMIS via the TkCoordinator instance. Two commands, **schedule_wakeups** and **cancel_wakeups**, are implemented. The former schedules timer wake-ups by registering the name of the wake-up procedure to be called and the polling period. The latter command is for cancelling the existing wake-up(s).

4. Making OSI Stack Independent

The string-based nature of Tcl can lead to different approaches in defining Tcl-MCMIS detaching the OSI stack. This observation has led to a fully string-based CMIS/P specification [LCMIP] that may be used locally over a pipe or remotely over a reliable transport service e.g. TCP, OSI TS etc. This may be used to construct a generic management access server which receives string CMIS messages, forwards them to the addressed remote application and returns the results/errors to the invoking client.

A number of Tcl CMIS extension instances may be communicating with the server from UNIX workstations, PCs etc. while the server runs on a UNIX system. This architecture results in OSI CMIP stack independence for the manager. For example, teleworkers with ISDN access in their homes can run Tcl CMIS clients on their PCs, communicating to a generic management access server whose front-end uses TCP, and the back-end performs requested operations on the target managed systems.

5. Conclusions

Tcl-MCMIS supports the easy construction of management tools such as MIB browsers, event monitors, etc. with minimal development time, compared to the compiled approach which involves a longer learning curve. Due to the interpreted nature, performance degradation, robustness and tolerance are obviously questionable. If a manager application involves heavy computations for realising the management intelligence, the policy concerned should better be implemented within C or C++, rather than processing in Tcl with the string operations. Large Tcl scripts can be very complex without showing clearly the program structure, and therefore they can be very difficult to maintain and debug! In order to avoid those pitfalls, a better approach will be to adopt the object-oriented methodologies supported by [incr Tcl] and Object Tcl with C++-like programming flavours.

Acknowledgements

We are grateful to Kevin McCarthy, of UCL, for his comments on an early draft. This paper describes work undertaken in the context of the RACE II Integrated Communications Management and the ESPRIT MIDAS projects. The RACE programme is partially funded by the Commission of the European Union.

References

[X701] ITU-T X.701, Information Technology - Open Systems Interconnection - Systems Management Overview, 7/91.

[X720] ITU-T X.720, Information Technology - Open Systems Interconnection - Structure of Management Information: Management Information Model, 1/92.

[X721] ITU-T X.721, Information Technology - Open Systems Interconnection - Structure of Management Information: Definition of Management Information, 2/92.

[X722] ITU-T X.722, Information Technology - Open Systems Interconnection - Structure of Management Information: Guidelines for the Definition of Managed Objects, 8/91.

[X710] ITU-T X.710, Information Technology - Open Systems Interconnection - Common Management Information Service Definition, Version 2, 7/91.

[X711] ITU-T X.711, Information Technology - Open Systems Interconnection - Common Management Information Protocol Specification, Version 2, 7/91.

[M3010] ITU-T M.3010, Principles for a Telecommunications Management Network

[Pav93] Pavlou, G., Implementing OSI Management, Tutorial presented at the 3rd IFIP/IEEE International Symposium on Integrated Network Management, April 1993, San Francisco, U.S.A. Also available as UCL Computer Science Research Note RN/94/80.

[Pav95] Pavlou, G., K. McCarthy, S. Bhatti, G. Knight, The OSIMIS Platform: Making OSI Management Simple, in Integrated Network Management IV, pp. 480-493, Chapman & Hall, 1995.

[Strou] Bjarne Stroustrup, The C++ Programming Language, Addison-Wesley, Reading, MA, 1986.

[Pav94] Pavlou, G., T. Tin, A. Carr, High-Level Access APIs in the OSIMIS TMN Platform: Harnessing and Hiding, in Towards a Pan-European Telecommunication Service Infrastructure - IS&N'94, pp. 219-230, Springer-Verlag, 1994.

[LCMIP] Pavlou, G., LCMIP: A Lightweight Protocol Architecture for Data and Telecommunication Network Management and Control, UCL Computer Science Research Note RN/95/61.

[TclTk] Ousterhout, J. K., Tcl and the Tk Toolkit, Addison-Wesley, Reading, MA, 1994.

[incr Tcl] McLennan, M. J., `[incr Tcl]` - Object-Oriented Programming in Tcl, AT&T Bell Laboratories, Allentown, PA, 1994.

Appendix

CMIS Error Codes and their Descriptions

Error code	M-Get	M-Set	M-Action	M-Create	M-Delete	Description
noError	.	.	.	.	.	no error has occurred.
noSuchObjectClass	.	.	.	.	.	the specified MO class was not recognised.
noSuchObjectInstance	.	.	.	.	.	the specified MO instance was not recognised. In M-Create, the specified parent MO instance was not recognised.
accessDenied	.	.	.	.	.	the specified MO instance cannot be accessed due to security reasons. In M-Create, the specified instance cannot be created.
classInstanceConflict	.	.	.	.	.	the specified MO does not belong to the specified class. In M-Create, the specified MO cannot be created as a member of the specified MO class.
processingFailure	.	.	.	.	.	a general failure occurred while processing the operation (usually out of memory).
invalidScope	.	.	.		.	the specified scope was invalid.
invalidFilter	.	.	.		.	the specified filter was invalid.
syncNotSupported	.	.	.		.	the specified (atomic) synchronisation is not supported.
complexityLimitation	.	.	.		.	one of the specified scope, filter or sync parameter was too complex.
noSuchAttribute	.	.		.		a specified attribute was not recognised. In M-Get and M-Set, this is an attribute level error.
getListError	.					some of the attributes could not be accessed due to errors. Partial results may be expected.
operationCancelled	.					the operation was cancelled.
setListError		.				some of the attributes could not be accessed due to errors. Partial results may be expected.
noSuchAction			.			the specified action type is not supported by the MO class.
noSuchArgument			.			the specified action information was not recognised.
invalidArgumentValue			.			the specified action information is invalid.
duplicateManagedObjectInstance				.		the specified MO instance already exists.

Error code	M-Get	M-Set	M-Action	M-Create	M-Delete	Description
noSuchReferenceObject				.		the specified reference MO instance does not exist.
invalidObjectInstance				.		the specified MO instance violates the naming rules.
missingAttributeValue				.		a required attribute value was not specified.
invalidAttributeValue				.		a specified attribute value was invalid.

An Example

The following sequence of commands shows the retrieval of the system object and its subordinates from an OSIMIS “SMA” agent running at host “kinou”:

```
% set agent [rmib_agent -a SMA -h kinou]
% set r [m_connect $agent]
% puts [m_get $agent -s baseTo1stLevel]
% set r [m_disconnect $agent]
```

yielding the following result:

1 1 SMA kinou M_GET noError

{noError {system} {} {19950714170258} {objectClass {system}} {nameBinding {dummy-OID}} {systemId {kinou}} {systemTitle {c=GB@o=UCL@ou=CS@cn=SMA@systemId=kinou}} {operationalState {enabled}} {usageState {idle}}}

{noError {subsystem} {subsystemId=4} {19950714170258} {objectClass {subsystem}} {nameBinding {subsystem-system}} {subsystemId {4}}}

{noError {uxObj1} {uxObjId=test} {19950714170259} {objectClass {uxObj1}} {nameBinding {uxObj1-system}} {uxObjId {test}} {sysTime {950714170259Z}} {wiseSaying {it’s easy with osimis-4.0}} {nUsers {1}}}

{noError {monitorMetric} {scannerId=uxObjId=test} {19950714170259} {objectClass {monitorMetric}} {nameBinding {scanner-system}} {scannerId {uxObjId=test}} {administrativeState {unlocked}} {granularityPeriod {secs:10}} {operationalState {enabled}} {observedObjectInstance {uxObjId=test}} {observedAttributeId {nUsers}} {derivedGauge {1}} {severityIndicatingGaugeThreshold {{Low:5 Switch:Off High:8 Switch:On}}} {severityIndicatingTideMarkMax {maximum: cur 1 prev 0.00 reset 19950714170212Z}} {severityIndicatingTideMarkMin {minimum: cur 1 prev 0.00 reset 19950714170212Z}} {previousScanCounterValue {0}} {previousScanGaugeValue {0}} {counterOrGaugeDifference {False}}}