

Master's Thesis in
Computing Science
Examensarbete DV3
2001 – 10 - 28
ISSN 1100-1836

The Managed Resource Interface: Interfacing Erlang with Standardized Management Protocols

Francesco Cesarini



UPPSALA UNIVERSITET

Computing Science Department
Uppsala University
Box 311
S-751 05 Uppsala
Sweden



Ericssons Utvecklings AB
Box 1505
S-126 25 Älvsjö
Sweden

Supervisor: Michael Williams
Examiner: Mikael Pettersson

Abstract

This report describes how different standardized management protocols can be incorporated into telecom applications with little or no impact on the overall software architecture. It presents a framework providing the necessary flexibility needed when different standardized management protocols have to coexist along side each other, or need to be interchanged due to changing system requirements.

The report shows there is a mapping from this framework to management protocols commonly used in telecom applications, namely SNMP, Corba and HTTP. OSI based management is also superficially discussed. The framework has been developed in Erlang, but can be used in any high level functional programming language.

Keywords: standardized management protocols, telecom applications, network management, Corba, SNMP, HTTP, Erlang

Contents

INTRODUCTION	1
CHOICE OF MANAGEMENT PROTOCOL.....	2
INTRODUCTION	2
OSI VS. IP.....	4
PROTOCOL DISTINCTIONS AND FEATURES.....	5
INFORMATION MODEL.....	6
TOOLS.....	7
SYSTEM REQUIREMENTS.....	7
MANAGEMENT SYSTEM REQUIREMENTS	8
SYSTEM FLEXIBILITY REQUIREMENTS	9
INTRODUCTION	9
PROTOCOL EVOLUTION.....	10
PROTOCOL ADDITIONS AND CHANGES.....	10
THE MANAGED RESOURCE INTERFACE.....	12
THE MANAGED RESOURCE INTERFACE	14
INTRODUCTION	14
THE COMMUNICATION MODEL.....	14
THE INFORMATION MODEL – <i>FUNCTION DEFINITION</i>	16
<i>Getting Data</i>	16
<i>Setting Data</i>	17
<i>Notification of events</i>	18
THE INFORMATION MODEL – <i>DATA DEFINITION</i>	19
<i>Defining records</i>	19
<i>Defining Actions</i>	20
<i>Alarms & Other Events</i>	21
THE ORGANIZATION MODEL	21
THE FUNCTIONAL MODEL	21
MANAGEMENT PROTOCOL MAPPING	23
SNMP BASED MANAGEMENT	23
<i>Functional</i>	23
<i>Communication</i>	24
<i>Organization</i>	24
<i>Information</i>	24
<i>Defining Managed Objects</i>	25
<i>Operations on Managed Objects</i>	26
WEB BASED MANAGEMENT	28
<i>Communication model - HTTP</i>	28
<i>Information Model</i>	29
CORBA BASED MANAGEMENT	30
<i>Organization and Communication Model</i>	30
<i>Function Model</i>	30
<i>Information Model</i>	31
OSI BASED MANAGEMENT	34
<i>Information and Communication Models – CMIP / CMIS</i>	34
<i>Functional model</i>	35
ANALYSIS	37
THE COEXISTENCE OF THE MRI	37
THE MRI AS A PROPRIETARY PROTOCOL	38
GENERIC UTILITIES	39

<i>Bridging and Interfacing</i>	39
<i>Generic Tools</i>	40
DESIGN FLAWS	40
<i>MRI operations</i>	40
<i>Erlang</i>	41
CONCLUSION	42
BIBLIOGRAPHY AND REFERENCES	44
ACKNOWLEDGMENTS	46
APPENDIX A: ERLANG	47
LANGUAGE ABSTRACT	47
DATA OBJECTS	48
FUNCTIONS AND MODULES	49
PATTERN MATCHING.....	49
FLOW OF CONTROL	49
CONCURRENCY	50
DISTRIBUTION.....	51
AUTHENTICATION.....	52
OPEN TELECOM PLATFORM.....	52
DESIGN TECHNIQUES AND METHODOLOGY	53
APPENDIX B: MRI EXAMPLE	54
THE EQUIPMENT MODEL.....	54
THE EVENT AND ALARM MODEL	57
APPENDIX C: ABBREVIATIONS	59

Introduction

Standardized management protocols have played a major role in the software architecture of telecommunication systems. They provide an interface for remote management applications developed by different vendors. Vendors and standardization committees put great effort and large resources into the development of these protocols.

An information model is an abstract representation of the managed resource. Within the domain of standardized protocols, this model is represented with syntax and semantic rules, providing a set of data structures and operations. Unfortunately, this information model is not available in the initial system development phases, but is the result of an independent standardization committee. This leads to evolving information models that often require major software redesigns.

Designers have a wide choice of protocols, where each has its strengths and weaknesses. This choice often influences the architecture of the system, yielding a tight connection between the representation and the internal structure. Such a connection results in inflexibility, making the system hard to adapt to changing market requirements.

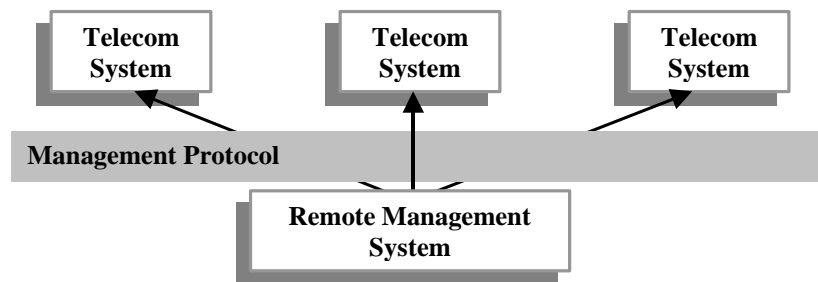


Diagram 1: *Using Management Protocols to control remote telecom networks.*

This report describes a framework that clearly separates the external standardized protocols and the internal data representation. The framework also provides an application-programming interface (API) that can either be used on a stand-alone basis as a proprietary management protocol, or as a base on which standardized protocols and information models can be built. The API includes organization, function and information models that use Erlang syntax and semantics. It also covers the communication model, based on Erlang distribution principles.

The report shows that there is a transformation of the API to standardized protocols and information models used today. This mapping is abstracted to include a generalized description on how syntax and semantics can be mapped to the major standardized protocols used by the telecom industry. A special emphasis is placed on SNMP. Other protocols such as CORBA, CMIP/CMIS and HTTP are superficially examined.

Choice of Management Protocol

This chapter analyzes the problems involved in choosing which standardized protocol to adopt between a management system and its network elements. It covers factors to be considered when making such a choice. Readers familiar with telecom networks and standardized management protocols can skip this chapter.

Introduction

Telecom networks consist of the infrastructure providing the transfer of voice and data between peer applications. The end applications will be connected through an access network. Fixed systems are referred to as wireline access networks and include standards providing services such as analogue telephony (PSTN), telephony and data transfer (ISDN) and packet based broadband services (ADSL). Radio access networks will provide the infrastructure to handle analogue and digital telephony such as NMT, GSM, CDMA alongside packet based GPRS and UMTS services. Connecting the various access networks together is a backbone transport network consisting of routers and switches transporting data using protocols such as IP, Frame Relay and ATM. Connected to the backbone and the access networks are a set of centralized servers providing services to the applications. The wireline and radio access systems, the switches and routers in the backbone, and all the servers supporting these services are referred to as network elements (NE). Network elements are controlled through remote network management systems, using management protocols to manage and supervise this infrastructure. This thesis will go in depth with the application level of the management protocols, and only briefly look at the transportation level.

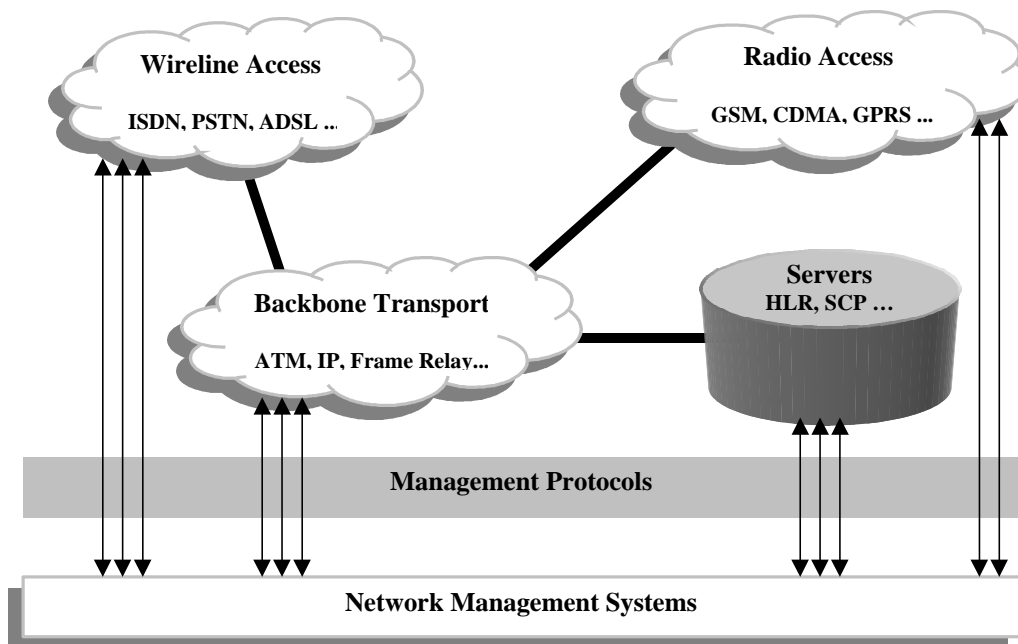


Diagram 2: An abstract telecom system.

Management protocols provide standardized means of describing management specific aspects of telecom applications. They contain syntax and semantic rules, or references to such, which allow a detailed description of the network element being managed. The Interface Definition Language (IDL) is for example used for defining objects managed through Common Object Request Broker Architecture (CORBA). A subset of the Abstract Syntax Notation 1 (ASN.1) is used to define elements used in the Simple Network Management Protocol (SNMP). Alongside with a standard of describing these objects, often called Management Information Bases (MIBs), other models define the communication properties and the supported functionality of the protocol.

There is no one ideal standardized management protocol. Had this been the case, there wouldn't be any need for the great efforts and large resources set-aside by the industry to develop these protocols. These efforts and resources have resulted in a vast choice of standardized protocols, with vendors offering a wide range of tools and platforms based on them. These tools and platforms greatly facilitate the development and integration of the management systems and network elements based on these protocols.

The choice of protocol is usually a trade off between complexity and simplicity. On one end of the scale, there are protocols whose aim is simplicity. They were developed to ensure easy agent implementation and short turn around time in software development. These protocols are better structured and easy to understand. Their simplicity, however, often requires extra functionality having to be inserted in the mapping of the information model, thus increasing its complexity.

On the other hand, there are protocols developed explicitly for the operation, administration, maintenance and provisioning of telecommunication systems. They contain a wide range of operations resulting in complex but powerful agents. The trades off in using these protocols are simpler applications that are easier to develop, shifting the complexity on to the agents.

An assessment of the strengths and weaknesses of each protocol candidate has to be made within the domain of the application. There may be scenarios where different management protocols should be combined within the same network element yielding the most powerful solution.

Management paradigms are under constant evolution. Early management systems were simple ASCII based ones, where a lot of the logic was placed in the network element. As this logic was slowly moved to the management system, proprietary management protocols were used for communication between the peers. They were later replaced by simple standardized management protocols such as SNMP, where more of the complexity and processing was moved to the manager. SNMP was intended as a transient protocol, and there were plans to replace it by the Common Management Information Protocol (CMIP), based on the work by the International Standardization Organization on Open System Interconnection (ISO/OSI). After a slow start, when products and tool kits finally had become available, so much had been invested in SNMP that migration plans had to be abandoned. Furthermore,

CMIP proved to be far too complex both as regards to agents and management systems. Resources were instead concentrated on the coexistence of the two.

After CMIP, CORBA gained popularity. This popularity was due to its abstraction, hiding CMIP's complexity and introducing communication transparency. SNMP has also continued to grow, and there seems to be a tendency in network management to migrate back to simpler protocols and managers. Proprietary web based management has for example gained wide acceptance, opening for a comparison to the early ASCII based systems.

Efforts were originally placed on developing new management paradigms. They have now shifted to facilitating the coexistence of these paradigms. This provides a uniform integration of old and new management systems and NEs.

OSI vs. IP

When discussing open network management, two vendor independent approaches are taken. They are referred to as Open System Interconnection (OSI) based management and the Internet Protocol (IP) based management. Both approaches describe the inter-process communication models, which in turn affect several aspects of the standardized management protocol definitions.

The OSI reference model defines 7 layers that serve as a basis for the specification of services and protocols. The layers include the physical, data link, network, transport, session, presentation and application layers. Interaction between entities of adjacent layers is strictly controlled, being based on interfaces, but horizontal communication between layers of the same level is open. All errors are for example handled horizontally among the peers, not involving the other levels.

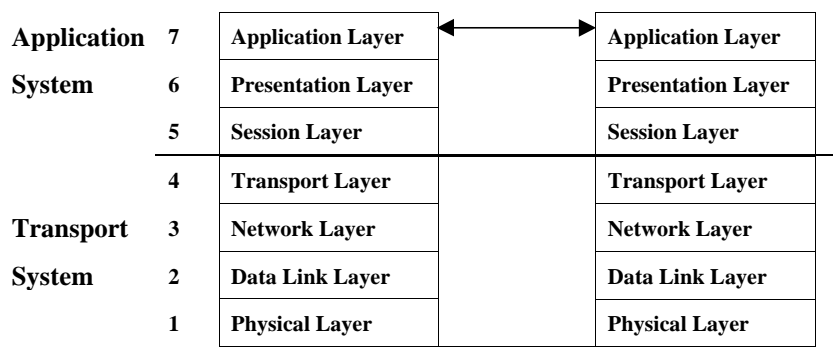


Diagram 3: *The ISO layer model, where all the different levels on the peers communicate with each other.*

The OSI model achieved wide relevance when the International Communication Union (ITU-T) started using the concepts as a basis for the Telecommunication Management Network (TMN) architecture. This acceptance was however greatly

delayed due to political reasons (see [Rose, 1994]), giving IP based network management a new renaissance.

The IP based protocols originated in the 1970s with the ARPA network. Their breakthrough came when they made a component of Berkeley UNIX. Standards are agreed on by the Internet Activities Board (IAB) through request for comments (RFCs), which are agreed or rejected by the committee. Layers 1 – 4 map to the OSI model, while layers 5 – 6 are encapsulated by application specific protocols based on Transmission Control Protocol (TCP) for connection based protocols and the User Datagram Protocol (UDP) for connectionless based protocols. TCP and UDP are in turn based on IP. Examples of TCP and UDP based protocols described in RFCs include TELNET (RFC 854), the Simple Mail Transport Protocol (SMTP), SNMP and HTTP. For a complete list, see [WEB].

The main difference between OSI and IP is that every RFC has to be submitted together with a prototype implementation. This ensures that the RFC specifications will not specify hard to implement features. This does not apply to OSI, resulting in the concurrence often being true for such standards.

Protocol Distinctions and Features

When comparing management protocols, [Hegering and Abeck, 1994] divided up their properties into four types of features. They include the communicational, informational, functional, and organizational models. The properties of these models should be considered alongside the acceptance of the protocol in the domain of the application, and the availability of tools and platforms supporting it.

The communication model deals with the exchange of information across the network. It looks at issues such as scope, reliability and encoding. Reliability can for example include facilities for detection of lost datagrams, and issues on where this functionality is to be inserted if it is not available through the protocol. What is the size of the datagrams in relation to the data exchanged, and is the protocol using a connection based or a connectionless underlying protocol? What type of encoding is used, and how does it affect efficiency are other issues of major importance which should be considered.

The information model deals with the description of the resources to be managed. What type of methodology is connected to the protocol? Is it intended to be used within an object oriented paradigm, a variable based one, or none at all? It looks at the state model for the entities represented how they can be described, and what functions may be applied on them. Most important in the information model is how the physical and logical abstraction is achieved.

The functional model specifies the functionality supported by the protocol. It includes means to handle the lifetime cycle of the managed entity as well as functionality to support specifications for testing, administration and maintenance of the entity. Some protocols have a minimal functional model, where the complexity is instead relocated to the logical description of the managed entity in the information model. The major

areas covered within the telecom world include the provisioning, performance and fault management, accounting and billing, as well as security aspects.

The organizational model describes the different roles of the systems involved, namely the manager agent paradigms together with the variations allowed by the protocols. The model may include proxies and agents acting as managers, and the hierarchy created thereof.

There is a growing tendency to use simple protocols in systems which until recently were managed by more complex ones. This also applies to applications where existing network elements already support these protocols. Mapping technologies between protocols have been developed and reached the maturity of allowing management systems to handle a diversity of network elements using a variety of protocols. [Clemm, 1996], when referring to SNMP, believes that another reason is the rich variety of generic tools available for simpler protocols. Such tools would be too hard to develop for more complex protocols should their generic properties be maintained.

Information Model

The choice of protocol is directly correlated to the choice of information model. Some protocols have no explicit information models. The information model is integrated in their function, representation and communication models. Other models represent the managed entity in terms of variables grouped together, abstracting away from the physical world. Object oriented paradigms are also used to logically abstract the network element, specifying inheritance and containment properties between the managed objects.

There are two types of information models. The proprietary models are vendor specific interfaces. The standardized models are agreed on and adapted by the industry. In early stages of development, and possibly in the first versions of the product, information models have not been standardized, and are under constant evolution. Inconsistencies can be found and fixed or functionality not supported by the protocol may be added.

The software has to be designed in a way that easily allows these changes. Whenever possible, the information model should as closely as possible relate to the internal state and data representation of the managed entities in the software. Attempts, however, are often made at early stages of the development to create abstractions and local optimizations of the model, thus affecting the software architecture. Abstractions and protocol specific optimizations should occur only when the product evolution has stabilized, or when the information model is replaced with a standardized one.

The need of standardization is expressed by the effort made by the industry to define the management information to be exchanged. Standardized information models relating to specific protocol leads to openness, allowing products developed by different vendors to be combined or interchanged with each other. All commonly used standardized management protocols have standardized MIBs, but it is common that MIBs that are standardized for one protocol have not been standardized for another. In

order to use tools developed for standardized MIBs within the area of the product, the product might have to support more than one protocol.

The Internet Activities Board has several working groups providing standardized MIBs on network technologies and components. These MIBs are often based on other existing standards derived by organizations such as IEEE or ETSI. There are also specific organizations such as the ATM forum, which define MIB definitions for devices supporting the Asynchronous Transfer Modes, or the ADSL forum, which, define MIBs for devices supporting Asymmetric Digital Subscriber Lines.

Whenever a standard appears in the area of the application of the managed entity, it should be supported on the network element side in order to allow the usage of tools and management systems based on these standards. This is however not the same as saying that the management system being developed for this particular resource must be adapted to follow the standard. It is possible that the proprietary information model available is more effective towards the specific network element implementation, and as long as there are no plans on using the management system on managed entities developed by other vendors, there should be no reason to change it.

Tools

The choice of tools available for a specific standardized protocol plays a major role in the choice of the protocol to deploy within the system. According to [Hegering and Abeck, 1994], tools written to coexist with standardized management protocols support three tasks. They are data collection, data reduction and analysis, and performance prediction. The wide range of tools available for network elements supporting SNMP was one of the major factors contributing to SNMP's success. If a network device supports a standardized SNMP MIBs, there will almost certainly be a ready built management system to handle them. Would the same apply to other management protocols?

It is possible that customers have a legacy system for accounting and billing and want to continue using it with newly acquired products offering new services. It is also possible that they have other tools for performance monitoring and statistical analysis that can be reused if the new device supports the specific protocol. Often, these tools provide a base on which the proprietary aspects of the management system can be developed and used alongside the already provided standardized one. Another factor influencing are the existing tools providing functionality in the requirements that would be cheaper to buy or license than to develop from scratch. Tools are also temporarily used awaiting more complex ones that are developed within the project.

System requirements

General system requirements contain issues that are often referred to and solved in the requirement specification of the product being developed. If the technology being implemented is young and on the rise, requirements are bound to change during the development phase. Customer requirements might affect the system specification as well. Would it in that case be wise to wait with the insertion of the standardized management protocol until the requirements can be considered stable, replacing it

with a proprietary interface? Or is there a protocol which when used and changed minimizes impacts to the already developed software, and possibly provide the designers with the needed flexibility?

There are cases where a new product has to be adapted to the individual needs of a readily existing management system. This would thus involve the ability of implementing the NE specifics in accordance with proprietary information models supplied by the customer or provide a mapping with the protocol and information model used. Is the standardized protocol being chosen the right one to provide this flexibility?

Management System Requirements

What are the typical requirements of a management system? Just like in standardized protocols, they can be divided into communicational, informational, functional, and organizational requirements. These requirements, however, do not implicitly deal with the underlying implementation. They focus more on how the management system is perceived by the operator using it. The choice of standardized protocol will directly affect the implementation of these requirements.

The communicational requirements consider response time, security on open networks, and ease and reliability to communicate with the managed entities. Functional requirements deal with what operations are available, and the ease of applying these operations. Will the system automatically attempt to fill in default values or derive values not entered, or must every parameter be specified? Informational requirements involves the needs of how feedback on the system is displayed, how performance monitoring graphs are set up, or on how detailed the information on the states of the managed entity has to be. The organizational model concerns how the whole system is displayed to the user, either in its physical or logical representations.

System Flexibility Requirements

The chapter examines how the choice of standardized management protocols can influence and limit the evolution of the system by negatively impacting the software architecture. The concept of a Managed Resource Interface in the network element, eliminating the influence and design flaws caused by the protocols, is introduced.

Introduction

If the standardized protocol is badly separated from the rest of the system, there is a risk that it will heavily influence the software architecture on both in the network element and the manager.

This influence in the architecture often occurs when attempts to optimize the application to a specific protocol are made. These optimizations consist in the attempting to reduce the size of the application and increase its efficiency. Other influences are created when inconsistencies are found and fixed in the wrong subsystem, creating a work around instead of solving the problem at its root. The standardized protocols should be placed on top of the managed entities, and not be incorporated into them. In nearly all cases, modeling an application around a management protocol results in an inefficient system.

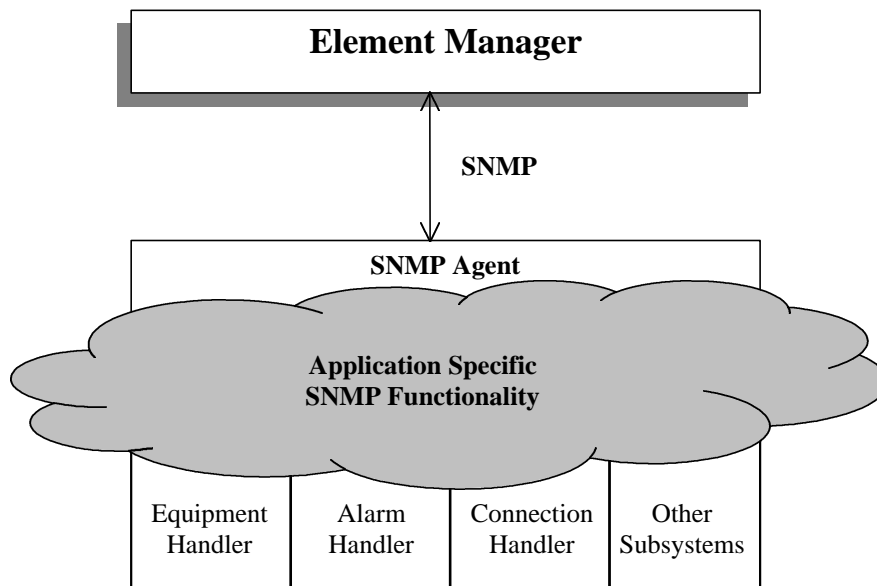


Diagram 4: ANx HFC provides broadband services through coaxial cables. In its software architecture, the application specific SNMP code and the Central Processor (CP) subsystems are so tightly coupled to each other that the introduction of a new standardized management protocol was not possible.

An example of standardized management protocols influencing the software architecture is a set of counters found in the UNIX BSD kernel. These counters keep track of sent IP packets, and have thus little to do with the kernel's central functionality. They were inserted in the kernel after the SNMP-V2 MIB was standardized, as the MIB required these counters. Placing them in the kernel was probably the easiest way of complying with the standard, thus resulting in non-core functionality being inserted in a core subsystem.

This independence in the software architecture should be achieved by adding a generic layer in the network element and in the manager. The goal of this layer is to separate the software architecture from the specifics of standardized protocols. Should requirements demand that a standardized protocol be added or removed, this can be done by changing this additional layer and not influencing the other parts of the system.

Protocol evolution

New standardized protocols are being developed all the time. Some become popular and are used in a wide range of areas, while others are used only within the specific niche they were originally created for. When is it acceptable to choose a new standardized protocol instead of some older proven one when both can handle the same requirements?

What tools can be used to facilitate the development of the manager and network agent for a specific application if the new protocol is used? What extra functionality can be offered to the end user which is not included in the requirements, but which is integrated with little or no extra work due to the protocol choice?

Common sense and extensive prototyping should prevail in the choice of a new paradigm instead of just following the flow and fashion of the moment. It is possible that requirements can be met and fulfilled with older models where tools and competence is readily available, resulting in the same effort. This alternative protocol might contain functionality that is needed due to changing market requirements, or be hard to implement in the framework of the existing implementation.

Whatever such a decision is based on, it is important to be able to backtrack and change or add standardized management protocols. This implies that the network element and manager software have to be implemented in such a way that a change of protocol, at any one time of the development stage, will minimize impacts on the system.

Protocol Additions and Changes

There are several solutions to the problem of having to introduce a new protocol in a network element. The most inflexible method, called a multi-protocol management agent, is adding one protocol's capabilities to another. Such a method is rarely used, and not recommended due to its complexity and need to re-implement large parts of the information model.

Another possibility is a multi-protocol manager, where the actual standardized protocol used is hidden in a layer within the manager. This layer offers an internal homogeneous model to the management applications, hiding the standardized management protocol differences. This requires management platforms equipped with the possibility of adding these internal extensions.

The most commonly used solution consists of a management gateway, also called bridging. This gateway can be placed in the management system, in the network element, or in a mediation device between the two. It maps one protocol to another. This method poses the question of being able to use the full capability of the agents provided. When mapping a less powerful protocol to a powerful one, little effort is needed, as the gateway does not need to fill any functional gap between the two. It is enough keeping track of the addressing schemes. It is however harder when a powerful protocol is to be mapped towards a less powerful one. When mapping a powerful protocol towards a less powerful one, the functionality has to be implemented in the gateway.

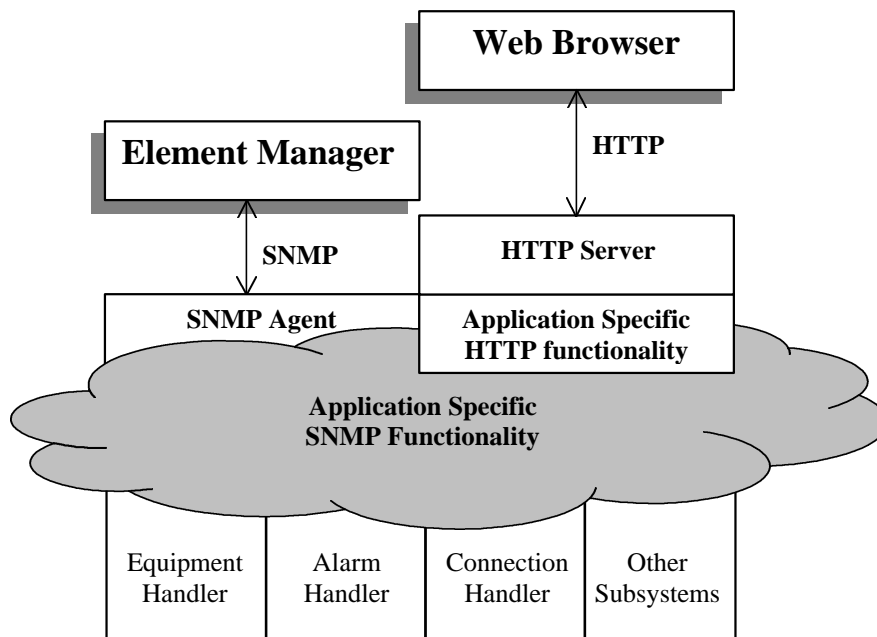


Diagram 5: The AXD is a multi-service ATM switch. In its software architecture, application specific SNMP code is too tightly coupled to the CP code. The insertion of a new management protocol has to go through the SNMP instrumentation functions, using an HTTP-SNMP bridge.

This paper introduces a fourth method that builds on the idea of the multi-protocol managers, with the difference being that the generic layer is placed in the network

element instead of in the management system. This layer is achieved through a generic interface, treated as a proprietary information model. Through a simple gateway, the standardized management protocol agents use the interface to create, populate and manipulate their information models. This allows new standardized protocol and information models to be implemented and inserted alongside other existing management protocols and agents resulting in multi-protocol network elements. The interface can be further expanded with the communication, functional and organizational models so as to be used as a proprietary protocol towards management systems being developed alongside the network element.

The Managed Resource Interface

The Managed Resource Interface (MRI) portrays the network element as a relational database consisting of a set of predefined tables. The function calls provided are primitive and atomic, not allowing any implementation specific knowledge of the system to be used above this interface. Included is a set of generic function calls. They can be used to retrieve all the keys belonging to a specific table, or given a table name and a key, they retrieve just a single element. No function calls are provided to write in these tables. The only way to change the state of the network element, thus changing the values of the entries in these tables, is through a function call executing actions defined in the information model.

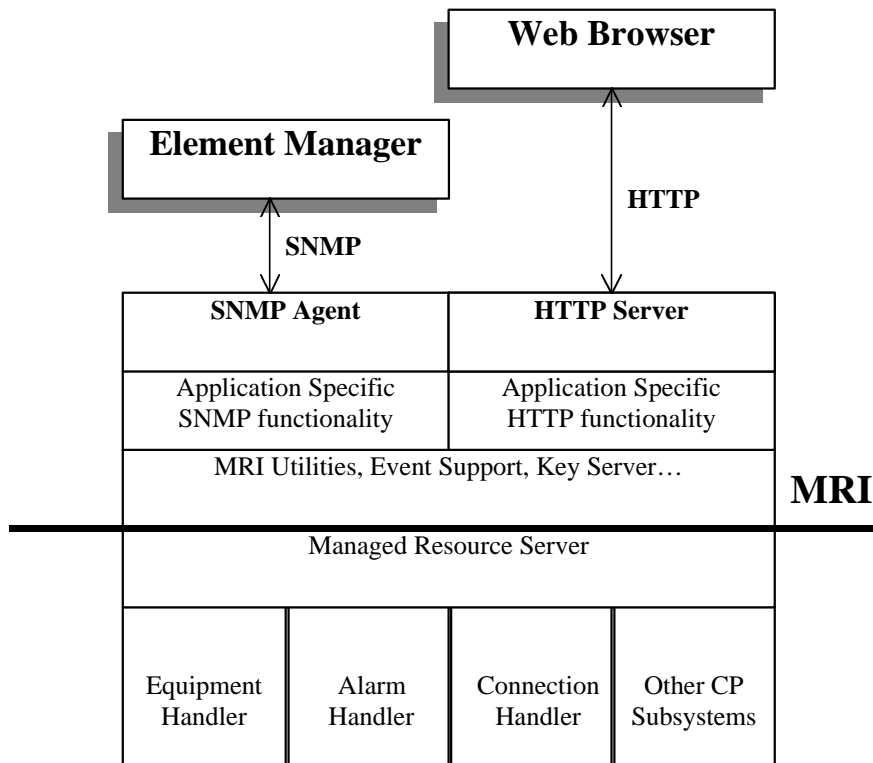


Diagram 6: The ANx DSL provides broadband services over copper wires. In its software architecture, the MRI is used to separate the standardized management protocol specific code from the CP applications.

The managed resource interface allows clients to subscribe to alarm and events from the network element. This is the only means of spontaneous communication from the network element towards the management system. Additional functionality for the subscribing and unsubscribing of events is provided.

By using the primitive functions in the managed resource interface, a set of generic tools can be built. These tools can export the needed functionality for a specific protocol bridging, and contain local optimizations. If built generically, these tools can be used regardless of the network element and the type of data exchanged.

An example of exported functions would be the implementation of the SNMP get-next operation, which is not exported in the MRI. Another operation could include the filtering of certain objects in relation to some defined attributes passed in the function call. These generic utilities should be developed once to facilitate bridging to standardized protocols in any network element, and should not be connected to any specific model.

The Managed Resource Interface

The following chapter introduces the Managed Resource Interface. It looks at the different aspects involved in the information, function, organization and communication models. It then gives an example over parts of the information and organization model for a typical telecom application. Readers not familiar with Erlang are recommended to read appendix A prior to continuing. An example of the MRI from a real project is included in Appendix B.

Introduction

The MRI is formed by a set of functions, generic to all applications. These functions take and return proprietary data structures that describe a specific managed entity within the application. The data structures, together with the generic functions, provide the infrastructure needed to manage the application.

The MRI attempts to portray the network element as a relational database where each table contains a set of record types. Each record type has a set of fields that represent a unique key in that table. All other fields are used to store specific data related to that instance. The record type, together with the key, represents a unique physical or logical managed entity.

The client using the MRI is allowed to retrieve information in two ways. It can either retrieve the complete list of keys associated to a specific table, or through a specific key, retrieve a specific instance of a record.

The client is only allowed to write elements that do not affect the logical state of the NE. All state changes of a logical element are achieved through a set of actions specific to that managed element. These actions influence the network element, resulting in a *create*, *write* or *delete* operation of the record describing the managed entity.

The MRI provides functionality that allows spontaneous messages originating from the network element to be forwarded to the clients. In order for clients to receive these notifications, they have to subscribe through the MRI providing a call back function.

The communication model

Should the MRI be used as a proprietary protocol directly towards the element manager, the specification for the Erlang distribution can act as a simple communication model. When standardized management protocols are placed as an extra layer on the MRI, no communication model is needed.

The model based on Erlang distribution would be a connection oriented IP protocol based on TCP/IP¹. The only security provided is based on Erlang cookies, where an Erlang node must be aware of other nodes' cookies in order to communicate. This

¹ Other protocols can be used if implemented, but no other implementation exists today.

might be suitable for networks behind firewalls², but in open networks, security extensions are needed. A possibility would be to incorporate encryption in the Erlang distribution model. The crypto library module provides such support.

If the distributed model is not extended, an extension has to be implemented on top of the MRI as an independent agent³. It could contain user session functionality, passwords, and access rights for different types of users.

Another possibility is using the secure socket layer provided with OTP. Socket and port communication would have to be used between the management system, often called the element manager (EM), and the processor controlling the network element, often called the Central Processor (CP). Communication servers would convert the Erlang terms to binaries, transferring them between the nodes, and converting them back to the original terms. Built-in functions for these operations are provided.

If the properties and security provided by the Erlang distribution is sufficient, the most common mean for an element manager to access the network element would be through the use of Erlang remote procedure calls. It would almost transparently allow the element manager to call the exported functions in the MRI, and receive the results as if the function had been executed on the Erlang node running the element manager⁴. The MRI data types could be used in the element manager, avoiding conversions and unnecessary layering.

```
rpc:call(CpNode, mr, get_keys, [#mr_current_alarm{ }]).
```

Example 1: *The remote procedure call syntax from the Element Manager to retrieve all keys of the current alarm list.*

Message passing between servers could also be used. It would however require an internal communication system (ICS) between the nodes. ICS could be implemented with generic server or finite state machine behaviors on the CP and EM sides, forwarding messages and keeping track over which links are up.

The Erlang distribution allows the monitoring of the CP Erlang nodes controlled by the EM Erlang node through the `monitor_node/2` BIF. The BIF creates notifications as soon as the communication between nodes is interrupted. The notification is generated regardless of if it is the TCP/IP connection that goes down, if it is the Erlang run time system which stops running due to a power shortage, a software crash⁵, or some other unforeseen cause.

² Messages are today sent unencrypted on the network.

³ The agent can run on the same Erlang system as the Central Processor.

⁴ This would also allow the element manager to execute on the same Erlang node as the CP.

⁵ A crash of the Erlang virtual machine, not the Erlang code executing.

Location transparency is another key feature, hiding the location of the CP in regards to the EM. As long as the CP is connected to the same network partition as the EM, no high level knowledge of the location or IP address is necessary⁶. The CP can operate regardless of the underlying hardware and operating systems, allowing hybrid systems using different operating systems to be connected together. Scalability can occur during runtime, and new CPs and EMs can be added without having to restart the system. No additions need to be made to allow several EMs to access one CP.

The Information model – *Function definition*

The functions provided within the information model of the MRI provide mechanisms for inspecting and changing the state of the managed entities. These functions are the most basic in the whole system, and it should not be possible to achieve the same state change through different set of operations. Complex operations and functionality specific to standardized protocols is built based on this model on top of the MRI.

Getting Data

The accessing of data is achieved through the `get_info/1` function call. The only parameter to this function is the record representing the table of the managed entity where the data has to be retrieved. In this managed entity record, only the keys need to be instantiated. All other fields are ignored. The CP extracts the keys and uses them to retrieve values that make up the MRI record, either from the CP databases, or by reading them from some hardware device. These values are formatted in accordance with the information model, returning a MRI record where all the fields are set. If this specific instance of the record we are trying to look up is not represented in the CP, an error tuple is returned.

```
mr:get_info(#record{Key1=K1,Key2=K2, ..}) -> {ok, Rec#record}|
                                             {error, Reason}
```

Example 2: *The syntax for the `get_info/1` call defined in the MRI.*

In order to read a specific record, the keys have to be retrieved. Such is done through the `get_keys/1` function call. This call takes an MRI record as a parameter, taking only the record type into account. All fields in the record are ignored. This function returns a list of lists. Each list consists of the keys defined in the records, positioned in the same order as they are defined.

```
mr:get_keys(#record{ }) -> {ok, [Keys]} | {error, type}
```

Example 3: *The syntax for the `get_keys/1` call defined in the MRI.*

⁶ Default network configuration of the underlying OS is sufficient.

Setting Data

Data that can be set through the MRI is divided into two groups, namely management data and logical CP data. Management data has no meaning to the CP, nor does it affect the CP's logical behavior. The CP just stores it in a table, providing a mapping to an existing MRI record. Secondly, there is data that affects the logical state of the CP when set through an action. The setting of such data often involves an operation or query on some hardware device, or some logical check of the data being passed, and can thus fail returning an error code other than the instance one.

Setting Management Data

Management systems are supposed to be stateless, and are thus not allowed to maintain their own tables. For this reason, all records defined in the information model of the MRI have a field called mgt. This field stores data that is specific for the managed entity denoted by the record and only needed and used by the management system. It might include strings displayed to the operator, identifiers denoting the physical location of the hardware, or possibly, standardized protocol information needed to send or retrieve other data.

```
mr:set(#record{key1=K1,key2=K2, ..., mgt = Mgt}) -> {ok, Rec#record}|
                                                {error, Reason}
```

Example 4: *The syntax for the set/1 call defined in the MRI.*

In the set operation, a MRI record in which only the keys and the mgt fields are set is passed to the call. These fields are extracted and stored by a subsystem in the CP. Their contents are never inspected. The return value of the function call is the MRI record where all the fields are set. Should the instance of the record not exist, an error tuple is returned. Whenever an action results in the record being deleted, the contents of the mgt field will also be deleted.

There are cases where a restart⁷ of the system is necessary, either due to software or hardware updates, or because of some bug or hardware malfunction. The information model describes which managed entities are considered persistent and need to survive such a restart. The contents of the management fields should also survive these restarts and be included in any future retrieval of the MRI records, without the data having to be reset. Management fields associated with volatile MRI records do not need to survive a restart, as these entities are recreated after the restart.

If the mgt field is never set, it will by default have the value of the atom undefined.

⁷ Often referred to as hot and cold restarts.

Actions

Actions in the MRI are the only means to create, delete or manipulate instances of records. These operations have to be atomic transactions, either succeeding and updating the internal CP state, or failing and leaving it untouched. Often, they involve some action towards existing hardware. To ensure the validity of this action, extensive logical checks on the current state of the system have to be made.

```
mr:action(Action, #record{key1=K1,key2=K2, ..., fieldN = Field}) ->
ok | {ok, Rec#record} | {error, Reason}
```

Example 5: *The syntax for the action/2 call defined in the MRI.*

The function call action takes two parameters. The first is an atom denoting the action and can be anything from *create* or *delete* to *configure* or *disable*. The second argument, an MRI record, denotes the managed entity the action is being applied on. The keys always have to be set. If one or more fields in the record are needed for the action, they are specified in the information model and set in the record being passed.

Notification of events

So far, the functionality provided has only dealt with the flow of information from the management system towards the CP. The MRI is also able to send notifications originating in the CP to the management system. These notifications usually involve alarms or events raised in the system, but can be used for other purposes as well. Any application in the Erlang node can subscribe to these notifications by giving the name of a call back module, a function, and a list of arguments. Whenever a notification is generated, all subscriber functions are called passing the notification as one of the arguments.

```
mr:subscribe(Module, Function, Arguments) -> ok.

calls Module:Function(Notification, Arguments) every time a notification is
generated.

mr:unsubscribe(Module, Function, Arguments) -> ok.

Removes the function from the notification list.
```

Example 6: *The syntax for the notification of events defined in the MRI.*

The Information Model – *data definition*

As the functions defined above will be the same in all systems using the MRI, regardless of the supported functionality. What will differentiate the various subsystems are the MRI definitions where the semantics of all system specific data models are defined.

The MRI model requires a detailed description of a set of records, each representing a managed entity, or part thereof. These records are then associated with a set of actions that, when executed on the managed entities, change their state.

Defining records

MRI records contain the data needed by the management system relative to a specific managed entity. All possible types that may be assigned to a field must be specified. They may contain any valid Erlang term, allowing related data to be grouped together in tuples, lists or other records. In the case of values consisting of integers, real numbers, or strings, it is usually sufficient to state the type, unless upper and lower bounds defined by some other standard exist. In the case of atoms, all combinations have to be specified. Process Identifiers and References, even if allowed, should be avoided and whenever possible replaced with counters and static process names. All type definitions of the record fields, as well as possible upper and lower bounds, are usually specified as Erlang comments in the include file where the records are defined.

A key in a MRI record may be contained in one or more fields. These are identified by setting the default value to the atom *key* in the record definition. A set of keys associated to a MRI record type must be unique and point out only one instance. Records without keys may exist, resulting in one such record instance on every CP.

It is common to describe the managed entities in a hierarchical fashion. A record will have a set of keys. The records on the next level will contain the same set of keys together with an extra one. It is also common to abstract wherever possible, creating a generic record describing different managed entities of a specific type. If these managed entities have properties that do not fit in the abstract record definition, extensions can be defined. Extensions may either be included as data structures in the record, or exist as an additional MRI record sharing the same keys and life spans.

The actual MRI records should not be stored in the CP. They should be instantiated only when needed, copying the data from other internal CP records or by reading it from hardware devices. When converting from one record type to another, a 1-1 mapping between the CP and the MRI records should where possible be kept. The CP record will contain the MRI specific fields, and in addition to those, other fields used only in the CP. These fields are ignored in the mapping, and never transferred. The same can be said about the layer above the MRI, where the MRI records are mapped to another record, extracting data stored in the management field. This mapping would probably be the internal format used by the element manager or by the

subsystem providing the translation to the standardized management protocol being used.

The correct definition of records is the key to successfully implementing a MRI. These records should follow the same principles used when defining and optimizing tables in a relational database. Data should not be replicated, and records should be divided up and optimized in regards to generic and specific data. If a mapping between primary and secondary keys exists, it is feasible to place it in an independent record defined for solely that purpose.

Defining Actions

MRI action definitions deal with changing the internal CP state, yielding a high-level definition of how the system should be configured and managed. The execution of these actions triggers events that change the state of the software and hardware. Actions may be general to a family of MRI records, or possibly be specific to a single one. Common generic actions include *create*, *delete*, and *configure*.

Action definitions specify which fields in the MRI records are to be used in the action. All other fields are ignored. Abnormal behaviors and side effects should also be documented. Abnormal behaviors might include switchovers, where standby equipment becomes active, and active equipment becomes standby. All of this is done in one operation, just sending a system node record as an argument to the action. The side effects, however, involve numerous boards in the active and standby tables. The records in which the hardware is represented change without an operation being applied directly on the managed instances.

Successful actions which are executed through the MRI either return the atom *ok*⁸ or a fully instantiated copy of the MRI record involved. Should an error occur, an error tuple is returned. Possible errors are defined together with the actions. Some error types are general to many managed entities, and could include *instance*, *instance_exists* or *hw_error*. Others are specific to certain managed objects, and are returned only within the subset of the MRI the actions are defined in.

Actions should be atomic, and no two actions dependent on each other should be combined. In telecom systems, an example would be setting the management status of a slot to unmanaged. This means that any hardware inserted or already in this slot will be ignored. Disabling a slot has the precondition that if a board is inserted in the slot, the board also has to be disabled. Boards can be enabled and disabled regardless on if the operator wants to disable a slot, and a slot's management status may be disabled regardless of if there is a board inserted or not. It would thus be incorrect to incorporate the disabling of the board and of the slot in one operation. The only documentation of such a dependency should be an error returned in case the board is still enabled.

Care should be taken in avoiding to insert fields in MRI records to determine the type of action. Assuming a MRI board record has a field denoting a state taking the values *active*, *inactive*, and *create*. Calling an action *set_state*, and then relying on the value

⁸ Used only if the action is delete, as no instance of the managed object will remaining in the CP.

of the field to create, or set board state to *active* or *inactive* is inappropriate⁹ because the state field of the record will never have the value create when it is retrieved. One should instead have two actions called *create* and *set_state*, or preferably go a step further and call the actions *create*, *activate* and *deactivate*, not relying on the record field at all. The MRI record would then tell what is being *created*, *activated* or *deactivated*, making the code more readable and the information model more straight forward.

Alarms & Other Events

The information model contains a list of all alarms and events supported by the system. These alarm definitions contain the originator type, the category and severity. This specification is needed to simplify the mapping from the MRI to other standardized management protocols, ensuring the correctness of the mapping.

The type system in Erlang allows for alarms and state changes to be treated in the same manner, the only difference being that an active alarm list is kept for alarms that have not been cleared. All alarms and state changes are encapsulated in the same notification record, and forwarded up to the subscribing functions. Alarms and events stored in the log, as well as active alarms use a unique index as a key, which in turn is retrieved with the `get_keys` operation.

The organization model

The MRI was built to follow a manager agent paradigm. This paradigm can be easily adapted to fulfill specifications required by other standardized management protocols. Some protocols have few and specific network elements, each dealing with their specific requirements. Such can easily be achieved by defining a specific information model for that specific functionality.

The management system will have the knowledge of which network element provides what functionality, and execute the specific MRI operations accordingly. In the case of homogeneous network elements, the solution is even simpler. Other functionality that can be implemented with the MRI is a central processor distributed on several Erlang nodes and boards, providing fault tolerance, reliability, and high availability.

The functional model

No functional model has been defined for the MRI. Defining functionality for provisioning, performance and fault management, accounting, billing and security in the model would have greatly complicated the mapping of the MRI to other standardized management protocols. The complexity belonging to the functional model is instead contained in the information model.

⁹ This is a common mistake done by people that have worked extensively with SNMP.

The above, however, does not imply that a model cannot exist. A MRI functional model can be defined in terms of MRI records and actions. MRI records and actions have in fact been reused in different systems, resulting in the first defacto functional model. [See appendix B, A Managed Resource Interface Example].

Management Protocol Mapping

The following chapter shows there is a mapping from the Managed Resource Interface to the most commonly used management protocols. SNMP is the most important and is thus covered in detail. CMIP, HTTP and CORBA are discussed from a general perspective.

SNMP based Management

The Simple Network Management Protocol (SNMPv1) [RFC1157] falls into the category of the Internet Management protocols. It is the product of the Internet Engineering Task Force (IETF)¹⁰ when the need for a standardized means to control large TCP-IP based networks came to light in the mid 1980s. Due to its simplicity, it gained wide acceptance, resulting in its support by many vendors. There were deficiencies and limitations, however, so in 1992, work was initiated on SNMPv2 (RFC1142-1452). This improvement in performance, security and ability to build hierarchical systems quickly made it one of the most commonly used and supported management protocols within IP based networks (telecom networks included). SNMPv2 has however resulted in a more complex protocol than originally thought, and claims have been made that the S in SNMP now stands for sophisticated instead of simple [Berkhout, 1994]. The specification for SNMP v3 and its coexistence with SNMP v1 and v2 has since been developed.

Here follows a brief introduction of SNMP, necessary to understand the mapping to the MRI. This paper concentrates on SNMP v2, as it was the version used to implement the MRI. A special emphasis is placed on the information model. For a complete documentation, see [Rose, 1994] and [McGinns, 1997].

Functional

SNMP has no operations covering issues discussed in the functional models. This is solved through standardized Management Information Bases (MIBs) supported by the managed entities and their agents. These MIBs define common information belonging to the managed heterogeneous resources, allowing them to be monitored and managed in a uniform fashion through the MIB.

The first standardized MIB, also considered the most important one, is the MIB-II [RFC 1213], mainly used for configuration fault and performance management. It has grouped together variables belonging to the major protocols such as IP, TCP, UDP, SNMP and more. Monitoring remote networks as a whole was a problem for SNMP, as it had been built to monitor single nodes. This deficiency was for solved by defining the Remote Network Monitoring MIB (RFC1271). Other standard MIBs include AppleTalk (RFC 1243), Mail monitoring (RFC 1566), ATM (RFC 1695) and

¹⁰ A self-organized group who makes technical contributions to the evolution and engineering of the Internet and its technology. It is the principal body developing new Internet standard specifications.

more. New standardized MIBs are being developed all the time, and an up to date list of RFCs can be obtained from IETF's web page (see [WEB]).

Communication

The communication model between the managers and the agents is through Protocol Data Units (PDUs), where messages are sent asynchronously. Different types of messages are sent within different SNMP PDUs. Multiple requests, as long as they are of the same message type, may however be incorporated in a PDU. There is no guarantee that PDUs reach their destination, making the manager responsible for taking necessary precautions and actions. No requirements exist for SNMP's underlying protocol. The connectionless User Datagram Protocol (UDP) is most commonly used, but there are RFCs defining how to operate it on top of other protocols such as Ethernet, TCP/IP and Apple Talk.

The message types include the *get*, *get-next* and *set* operations. *Get-bulk* request was introduced in SNMPv2 retrieving large amounts of data through a depth first search. The agent, in reply to the manager requests, sends a PDU containing the *response* message. *Traps* are the only message types that agents are able to send spontaneously.

The SNMP agent provided with OTP will handle all the coding and decoding of the PDUs. It will then map the requests to Erlang function call stubs returning data used for the response messages.

Organization

The organizational model is based on the manager agent paradigm. The manager sends requests to the agent receiving a response. Scalability and reliability is achieved by creating hierarchies, where agents can also act as managers towards other agents located at a lower level of the hierarchy.

This allows lower level managers, referred to as proxies, to handle administrative actions. They are often automated, relieving the load off other already busy high-level managers. The middle managers can in addition distribute work delegated from managers at a higher level of the hierarchy to other agents. Information hiding is also made possible, as is transport protocol independence.

Information

SNMP's information model consists of MIBs describing the managed objects. They are defined by using a subset of the ASN.1 [X208], an internationally standardized language for defining syntaxes. A MIB contains the description of all its variables and tables, values these variables and tables may take, together with the global names used to access these contents.

MIB variables are scalar variables of which there only exists one instance per MIB. Tables are two-dimensional, where single elements are themselves scalar variables. Values in tables have to be accessed one column at the time, as SNMP's only

processable unit is a scalar variable. Operation on complete rows such as the creation or deletion can be emulated through the use of extra variables.

Defining Managed Objects

A MIB is a module containing managed entities logically grouped together. A unique name is chosen for the module, followed by imports from other modules such as syntax types, object types, and trap types. These imports are followed by the definition of the object identifiers (OID), dependencies of these OIDs to other standardized MIBs, and references to the parent in the naming tree, followed by Manage Object (MO) definitions.

The object type macro allows for three types of objects to be defined. They are tables, rows, and leaf objects. Tables are made up of rows, and rows are in turn made up of leaf objects. Scalar variables, which are unique in every MIB, are also leaf objects. A leaf object is the smallest grouping of information.

Leaf objects are accessed through an OID, where every managed object has a globally unique name in the naming tree. OIDs are a sequence of hierarchically organized non-negative integers. Values are stored in the leaves of this tree. To facilitate usage, a textual name is associated with each sequence element of the OID.

An SNMP MIB would map to a set of MRI record definitions logically grouped together. The tables map to collections of MRI record types, while the table rows will map to instances of these records. The leaves of the naming tree would in turn map to the specific MRI record fields. A scalar variable may either denote a columnar entry of a row or single variables. Single variables could be stored in MRI records with no keys, or derived in an intermittent above the MRI. Finally, OIDs map to a MRI record type and its set of keys, and the column number to a field in the MRI record or record extension.

```
SYNTAX INTEGER (0..65535)
SYNTAX INTEGER (0..'ffff')
SYNTAX INTEGER {
    et155(1),
    qam(2),
    ucm(3),
    hfc_nt(4),
    ce(5) }
SYNTAX INTEGER (0..127)
```

Example 7: *The first two definitions are for normal integers, the second for enumerated types, and the third regards an integer bitstring.*

Leafs can be of the type *integer*, *octet string*, or *null*. Based on these basic data types, more complex data types are defined (or imported from other MIBs). The mapping of these data types to Erlang data types is straightforward. Fancier mappings can be achieved by mapping enumerated types to atoms. The OTP MIB compiler will instead

create include files with macros. These macros will be replaced with integers by the precompiler.

Octet strings, and all data types derived from them, are lists of bytes. They can easily be mapped to Erlang lists or binaries. They are intended in being binary data chunks. The length may be fixed, variable, and its values could be given upper and lower bounds. Special cases of strings include the *octet bitstring* used when the size of a bit string is not enough, a *display string*, consisting of printable ASCII characters and the *IP address*, a four byte octet string.

All definitions originate from an OCTETSTRING or an integer, and are then mapped to the existing Erlang data types. Erlang references and process identifiers should be avoided in the MRI, but if used, a mapping to a unique integer or octet stream would be possible.

Operations on Managed Objects

As described in the communication model, there are three operations that can be applied on a managed object. They are asynchronous request - response types sent in PDUs by the manager. The agent asynchronously responds to these operations. Traps are also spontaneously sent, and involve a notification of a change in state of a managed object.

The SNMP requests, upon being received by the agent, are mapped to function calls. These function calls may be executed concurrently, and are usually handled by spawning a process for every request. The mapping provided here is simple, and does not take into consideration efficiency aspects. If generic tools such as a key cache or filtering requests are implemented above the MRI, a more efficient mapping than the one described could be achieved. Adding these layers is a necessity in a large system, but can be considered redundant if the mapping is simple.

The *get* operation takes an Object Identifier as a parameter. The agent will respond by encapsulating the contents of the scalar variable in a response PDU. The scalar variable could be either a column in a row, in a specific table, or a simple variable. If the OID is erroneous, the *noSuchName* error code is returned. By grouping together several *get* operations, the retrieval of a whole row in a table can be simulated.

In the MRI, the *get* operation would have a 1-1 mapping to the *get_info* call. The SNMP OID is mapped to the MRI record keys, and the column number is mapped to one of the fields in either in the MR record or in the extension record. Some agents support the returning of a whole row in the PDU. The mapping thus can be extended to return several of the MRI record fields.

The *get-next* operation has by [Rose, 1994] been called the *powerful get-next* operation. The power lies in its ability to return the next (in lexicographical order) object instance in the MIB tree, together with its OID. The OID is often recursively used in the following *get-next* request. Tables are traversed vertically, taking one column at the time, and not horizontally, as one might expect. Several *get-next* operations may be encapsulated in a PDU, allowing different tables to be traversed simultaneously.

The *get-next* operation is based on the ability to efficiently pick out the next column and key of a scalar variable. As tables are traversed vertically, the SNMP agent would return the same field in all the records in the MRI table, together with the next key and the column number. When this column has been traversed, the column number would be increased, and the field denoting the next entry in the SEQUENCE would be traversed. If the Column value is 0, we are dealing with a scalar variable, and thus map it accordingly, possibly to a MRI record with no keys, returning it together with the first key of the next table in the MRI (mapping to the next table in the MIB). The last case is if there are no more tables defined, to which the *get-next* operation returns *genError*. The API provided by OTP's SNMP agent allow the retrieving of one row at the time, avoiding the inefficient vertical traversing of the tables otherwise required.

The set operation takes an OID and a value that are sent to the agent. The agent replies with a success or failure of the write action in the response PDU. Error types include *noSuchName*, where the OID is erroneous, *badValue*, where the type of the value sent is not the type defined in the MIB. If the variable has no read-write access value, *readOnly* is returned. All other errors are covered by the *genErr*. Returning the *noError* type denotes a successful operation. Grouping all the set requests in a PDU simulates setting of a whole table row. The only error types originating from the MRI include *noSuchName* ({error, instance}) and *genErr*. *genErr* stands for all other error types that can occur. Such an error will result in an entry in a SNMP table where the SNMP manager can the specific error type can be read.

Set operations are mapped to MRI actions. If a whole table row is set, the MRI must be able to either provide an atomic operation for setting all the values, or a transaction mechanism that allows roll back in case of some, but not all the necessary actions succeed.

Web Based Management

Web based management is rapidly gaining popularity as expensive platform dependent management systems are quickly being replaced by web browsers. Browsers use HTTP over TCP/IP when sending requests to web servers. These requests are converted to queries and actions executed in the CP. The results are formatted into HTML and sent back to the browser where they are displayed. With the help of Java script, functionality can be moved between the browser and the CP. The most common practice is to check for basic syntax errors in the browsers, pushing the remaining functionality in the CP.

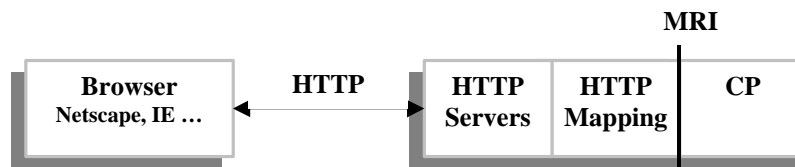


Diagram 7: *The MRI using HTTP over TCP/IP.*

Web based management has raised the issue of standardization, in that most vendors have developed proprietary protocols. The Web Based Enterprise Management (WBEM) under control of the Desktop Management Task Force (DMTF)¹¹, and the Java Management API (JMAPI)¹² are working on new standards. They are however still young, and according to [Oslebo, 1998], a lot of work remains before this new technology is accepted as a standard for network management. Concerns have also been raised over the fact that no work has been made over the integration of web based management standards with other existing protocols.

Communication model - HTTP

The communication model is covered by the Hyper Text Transfer Protocol (HTTP). HTTP provides an IP based connection where strings are exchanged. Most web servers available on the market provide log in security and encryption of the data being transferred.

These strings retrieved from the web server contain tags that often are a dialect of the Standardized General Mark-up Language, (SGML). The most commonly used dialect is the Hyper Text Mark-up Language (HTML), which provides means to graphically display data.

¹¹ Includes Microsoft, Intel, Cisco, Compaq and BMC.

¹² Includes Javasoft and its partners Cisco, Bay Networks, 3Com, and others.

Information Model

No defacto information model exists for HTTP, even if work is on the way. A simple mapping for web-based managers would today require an API, where tagged requests in some proprietary SGML dialect are mapped to instrumentation functions. The AXD 301 ATM switch, as can be seen in diagram 5, maps its HTTP functions to SNMP instrumentation functions. A more effective way, however, could have been mapping directly to the MRI, not having to take into account the SNMP specific behaviors.

Allowing a representation in the html code that is understood by the web server allows for higher-level actions to be mapped to the MRI actions. Representations in the web pages could include CGI-bin scripts, ehtml¹³ tags, calling or Erlang functions, or Java applets running threads towards the management system, interfacing Erlang on the CP. Simple type checks can be made using Java scripts or through a layer above the MRI.

HTTP is a connectionless protocol, so spontaneous notification of events from the CP is not possible. The simplest solution is by automatically reloading a frame with the alarm notification every few seconds. The second, and preferred way, is through a Java thread.

The organizational model would be a strict client server one, not allowing hierarchical systems. As the contents of the web pages will be created on the fly, no HTTP proxy servers should be used.

¹³ See User contributions of the Erlang Open Source project at <http://www.erlang.org>

Corba Based Management

The Common Object Request Broker Architecture is a standard derived by the Object Management Group (OMG) consortium¹⁴. Their mission was to write a specification for an Object Request Broker (ORB) providing a software bus that allowed objects written by different vendors to interact across networks. Corba based management uses an Object Oriented architecture where managed entities are defined in interfaces and accessed through client server relationships. It is more powerful than SNMP and less complex and easier to use than CMIP. Its affiliation with C++ and Java helped spread its popularity as these languages were given properties such as distribution, platform transparency and component based software modules. Corba 1.1 specifications, introduced in the early 1990s, specified the Interface Definition Language (IDL), language bindings and the interface towards the ORB. Programs could be written interfacing different ORBs. ORBs from different vendors, however, were only able to interact after the Corba 2.0 specification was released, as it described the inter-ORB operability across networks.

Organization and Communication Model

The organization model is a client server one, where servers can themselves take the role of clients. Clients can be seen as programs that invoke methods (or requests) on Corba objects (The EM), while servers are programs that implement these objects, performing the operations associated to them (The CP).

The communication model in Corba v 2.0 is based on the Internet Inter-ORB Protocol (IIOP). Different Corba implementations conforming to IIOP guarantee full interoperability with each other, facilitating inter-node communication in a distributed environment. Distribution is transparent, as clients invoking a service on an object do not need to know where on the network this object resides. Client requests on local objects have the same syntax as requests to objects on other nodes. The semantics differ, however, as local requests are handled within the ORB not using IIOP.

Function Model

The CORBA functional model consists of a set of services provided by the ORB. These services are themselves defined as CORBA objects, either by directly providing interfaces accessed by the clients, or by providing interfaces inherited by other object definitions. The most important services include naming, life cycle and event services. The life cycle service deals with the creation, deletion and relocation of Objects.

The naming service allows names within a naming context to be associated to an object. This association can later be used in resolving these names to Corba references, to create contextual object relationships (defining relationships between objects), or just to retrieve a list of objects bound to the naming context.

¹⁴ Includes 700 companies, Microsoft excluded. Microsoft has of course derived its own similar standard, DCOM.

The event service allows suppliers to provide consumers with events. Suppliers can send spontaneous notifications through the *push* model. Consumers can otherwise read events directly from the supplier through the *pull* model. In the *push-pull* model, suppliers generate the notifications, which are stored in an event channel (a queue) and eventually read by the consumer. In the *pull-push* model, both the supplier and the consumer are passive. The event channel will pull an event from the supplier and push it to the consumer.

As Corba is more complex than the MRI, the multi protocol management agent could be used to fill in the gap. This would imply that the MRI information model would have to be expanded to cover the functionality. The latter can be easily done for all Corba services not implemented in the ORB, but would defy the purpose of the MRI, as the information model should be developed independently of other management protocols. The best way of mapping Corba services to the MRI would be through bridging, where a software layer implements the extra functionality not existing in the MRI.

In the event service, for example, the push model has a one to one mapping to the MRI notification of events. All other the other models, however, would have to be implemented as an extra layer between the ORB and the MRI. No life cycle or naming service functionality exists in the MRI, but many of the dynamic objects have create and delete actions associated with them.

Information Model

Objects residing on a Corba bus are described as identifiable encapsulated entities providing services to the client. They are defined in a subset of the Interface Definition Language, a specification commonly used to define platform and language independent interfaces. These interfaces consist of unique object names, a set of functions, and the type these parameters and return values can take. They are compiled using a stand-alone IDL compiler, producing client and server code stubs.

The IDL function definitions in the interfaces are mapped to MRI actions and `get_info` functions. Return values of these functions can be simple or composite IDL data-types extracted from the MRI records. If a direct one to one mapping is to be achieved with the MRI interface, MRI records can at run time be converted to IDL structs. Every field in the MRI record is mapped to the respective field in the IDL struct. Error values returned from the MRI function calls are handled as exceptions on the client side, while actions returning the atom `ok` can in the MIB be defined to return void.

Table 1 maps the basic IDL data-types to Erlang data-types. Erlang does not support the double value range, so all conversions of a double will be mapped to an Erlang float. The data-type `any` is used in conjunction with type codes defined in the OMG Corba specification, allowing functions to be dynamically typed.

OMG IDL Type	Erlang Type
float	float
double	float
short	integer
unsigned short	integer
long	integer
unsigned long	integer
char	integer
boolean	'atoms <i>true</i> <i>false</i>
octet	integer
any	'record #any {typecode, value}
Object	Orber object Reference
void	'atom <i>ok</i>

Table 1: *The basic OMG IDL to Erlang type mapping.*

IDL objects are identifiers referring to other Corba objects. Corba objects are identified by unique references created by the ORB. These references are internal to the ORB, so a table has to be placed in a layer above the MRI storing the mapping of the references to the MRI keys.

The constructed OMG data-types are listed in table 2. They are self-explanatory.

OMG IDL Type	Erlang Type
string	list
struct	record
union	record
enum	atom
sequence	list
array	Tuple

Table 2: *The constructed OMG IDL to Erlang type mapping.*

The functions provided in the MRI should also be included in the Corba Object description. Corba requests take a target object, a request and a set of parameters returning an IDL data-type. MRI actions can thus be mapped to separate requests returning void (where the atom *ok* is otherwise returned) or the struct describing the managed entity for all other ones. Any errors can be raised as exceptions.

There should be a specific *get_info* function to read the struct, and if needed, a *set* one for the management data. Other specific requests not affecting the state of the CP can also be mapped to MRI set requests.

An operation similar to the SNMP *get-next* or *get-bulk* could be included in the object interface description mapping to the MRI *get_keys* function. The *get_keys* function would thus be broken down into several calls ensuring that the size of the data transmitted between the server and the client is manageable.

The Corba naming service can be useful when implementing the *get_next* or the *get-bulk* function. By binding the smallest (In lexicographical order) element to a static

name, it can be used to retrieve the first Corba reference. This reference is then used to read the remaining references until the end of the table is reached.

Interface inheritance is another property associated with the information model, where one Corba interface may inherit properties from other interfaces. Where defined, it can be assumed that the Corba objects inheriting properties from other Corba objects would in most cases map to MRI managed object hierarchies, which already share these properties in their MRI definitions. The MRI has no inheritance, so these inherited properties have to be defined on a managed object basis in the MRI. If that is not the case, the inherited properties would have to be defined on a managed object basis in the MRI.

OSI Based Management

The OSI management framework was derived by the International Organization for Standardization. Work was started around 1980, and the final publication [ISO 7498/4] was released in 1989. This framework was seen as inadequate, so work on an additional standard, the Systems Management Overview (SMO), was started in 1987. It was finalized in 1992 through the publication of [ISO 10040], which together with [ISO 7498/4] provides the standard for OSI management. When endorsed by ITU as a Telecommunication Management Network (TMN) standard, it achieved wide relevance. It is, however, by many considered to be complex, with many ambiguities remaining unresolved (see [Pras, 1995]).

OSI management uses an object-oriented paradigm, where the managed resources are abstracted and referred to as Managed Objects. Monitoring of MOs takes place through polling on behalf of the EM or through spontaneous events generating from the NE. Management operations deal with delegation of tasks and the distribution of intelligence among the MOs.

Exchange of management information takes place in three ways: systems, application, and layer management. In controlling MOs, however, one is usually able to narrow down the complexity and concentrate on the application layer¹⁵ as well as the information and communication models, both defined in the SMO. The MIBs of the information model are defined using a protocol called the Common Management Information Protocol (CMIP). Exchange of data using CMIP is achieved through the Connection Management Information Service (CMIS), which specifies the services provided.

The organization model consists of the manager-agent paradigm where the system, depending on the MO being handled, can take on any of these two roles during run time.

No Erlang CMIP / CMIS agent implementation exists today, but due to the relevance of the protocol in existing telecommunication systems, and the impact it has had on standardized management systems, it is included in this chapter.

Information and Communication Models – CMIP / CMIS

MO classes are grouped in MIBs, where the properties of the resources being managed are described. Properties include attributes belonging to the MO, notifications deriving from it, and actions that can be applied on it. The semantics of the MOs may be specified in the rules for containment and instantiation or they can be inherited from their super classes. Different levels of the inheritance hierarchy will then correspond to the different levels of abstraction. This abstraction allows for the reuse of these specifications in different application domains. MIBs are defined using ASN.1.

¹⁵ With layers 1 through 4 being handled by TCP/IP.

Managed Object Instances are arranged in a management information tree, which is expressed by a naming hierarchy. This naming hierarchy is used for scoping, which is the issuing of management operations to several MOs in the same sub-trees. MO classes, in addition, inherit properties from their super classes. When Inheritance applies to the MO classes, it should not be confused with the naming hierarchies that apply to the specific instances of these MO classes.

A MO description lists a set of attributes that are part of the object, characterizing its state and properties. These attributes will map directly to the fields in the MRI record definitions (possibly including data stored in the management field). The data types are similar to the ones described for SNMP containing both simple and composite data-structures, and thus need not be covered. It is worth noting, however, that there is no limitation to the complexity of the Managed Object Classes as there is with SNMP.

A set of actions is associated to a MO. They can be mapped directly to the ones defined in the MRI. The create and delete actions, handled separately in CMIP, could be mapped to other MRI actions whose execution results in the creation or deletion of the record instance. They could also be called *create* and *delete* in the MRI, even if it is not always the case.

Conditional packages are used to describe variants of certain managed objects. A managed object, depending on the value of one of its attributes, may trigger other types of MOs to be created. These variants are activated (or created) when the object is first instantiated. In MRI terms, there is no formal means to describe a variant, other than in the action description. The concepts can however be mapped to the MRI record extensions, which are either defined as a stand alone MRI record, or as a composite data-type stored in one of the MRI record fields.

The last feature belonging to CMIP MO definitions are the references to their super classes. There is no inheritance in the MRI, so the inheritance properties need to be redefined for all derived objects at all levels. It is however possible to see the dependency hierarchy of the system based on the key definitions, where the records on a level will contain at least the same set of keys of the previous level, and possibly an extra one.

CMIS allows clients to access and manipulate not only objects, but also whole containment trees. The primitives M-GET, M-SET, M-CREATE, M-DELETE, M-ACTION, and M-EVENT-REPORT are used to form the CMIP PDUs, which are exchanged among managers and agents. Using the get operation, it is in addition possible to filter objects in the containment tree, and by using scoping, actions may be applied to the set of filtered MOs which meet certain criteria. Tasks such as filtering and executing on the containment tree could be placed in an abstract layer right above the MRI.

Functional model

The TMN functional model covers areas such as configuration management, fault management, performance management, accounting and security. This model mentions CMIP as the protocol to be used, and includes a set of standardized MIBs containing generic services and uniform management interfaces. These MIBs, just

like the SNMP functional model, have to be mapped to existing MRI records and actions. Due to the complexity of CMIP, however, some parts of the functional model could be placed in a layer in-between the CMIP instrumentation functions and the MRI.

One of the services in this model includes the logging of the alarms and events generated by the MOs. The functionality for collecting and logging alarms could be placed under the MRI, as can be seen in Appendix B. Alarm handling is generic, and common to most applications, so it would be natural to have an MRI definition for it.

More complex functionality such as test management, however, could in large part be placed above the MRI, using only MRI primitives. Test management includes functionality for scheduling, starting and stopping tests, calculating and analyzing results. The MRI could export actions for starting and stopping the tests, where the return value of the stopping of a test returns possible results. All other functionality for scheduling and analyzing could then be placed above the MRI.

A similar analogy could be made for the polling of data. If the MRI has a record containing the data to be polled, the server that executes the `get_info` call at regular intervals (and possibly does the statistical calculations) could easily be placed above the MRI, thus minimizing the software impacts under the MRI.

Analysis

This section analysis the MRI. It looks at how it can interface in with the standardised management protocols discussed in this thesis, and how it can be used on a stand alone basis. It concludes by looking at some of the problems which were experienced in the existing implementations.

The Managed Resource Interface was first implemented in the ANX DSL project as a means of removing the SNMP dependencies in the logical applications of the CP, facilitating the insertion of new standardised management protocols. This was a major rewrite of the existing system, the ANX HFC, involving several geographically separated teams of engineers. The MRI was later used in the AxM3 project, with the MRI going into the design of the system from the start. In both these projects, SNMP and HTTP were interfaced towards the MRI. The analysis in this section is based on these project experiences.

The Coexistence of the MRI

Standardized management protocols differ in terms of power. Simple protocols such as HTTP or SNMP result in simple agents but complex applications. On the other end of the scale, complex protocols such as CMIP and Corba have complex agents but simple applications. It is a trade off on where the complexity is placed, namely in the data definition of the information model or in the functions available to access and manipulate this data. The Management Resource Interface can be placed in the middle of the discussed protocols, both in terms of data definition and complexity of the functions that can be applied on this data.

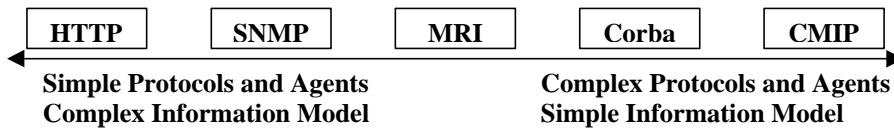


Diagram 8: *The complexity of the standardized management protocol and its information model.*

SNMP's information model consists of a complex information model with simple functions to manipulate it. This pattern is even more evident in the case of HTTP. Corba and CMIP, on the other hand, have simple information models, but more complex functions to access and manipulate the data, making the protocols more powerful.

This means that the mapping from SNMP and HTTP to the MRI is a simple task. There is no functional gap that has to be filled, as the functions in the MRI are more powerful. There is only a semantic gap mapping the HTTP and SNMP requests to MRI records and operations.

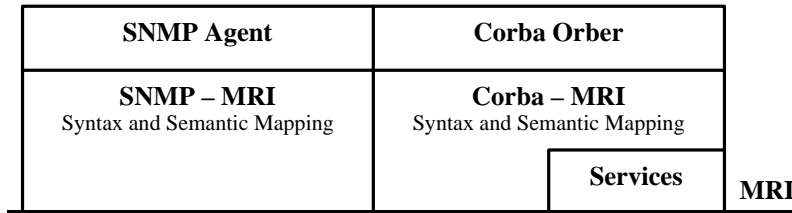


Diagram 9: *The mapping of the MRI to SNMP is a pure syntax and semantic mapping. Corba, however, contains functionality not included in the MRI that has to be implemented in a bridge separating the two protocols.*

When mapping Corba and CMIP, however, functionality not available in the MRI has to be implemented. This functional gap includes transaction, event, notification and time services, all of which exist in the Erlang/OTP Corba agent. This mapping can easily be implemented by inserting a bridge that handles the functional gap. The semantic translation can then be placed as a layer between the bridge and the MRI, or as shown in diagram 5, right above the MRI. If such a bridge was needed (It depends on what Corba services are used), with a little care it could be implemented generically allowing its re-usage among projects. No Erlang – CMIP implementation exists today, but a bridge would follow a similar principle as the Corba one, only that it would be much more complex, as there is a larger functional gap which has to be filled.

The MRI as a Proprietary Protocol

If an Element Manager is implemented in Erlang, the MRI should be used as a proprietary management protocol. Using communication protocols other than distributed Erlang adds an unnecessary factor of complexity to the software, penalizing the performance due to overheads such as the encoding and decoding of data.

If distributed Erlang is used, an extra layer of functionality should be included above the MRI. Issues such as node monitoring, and network optimizations, user authentication and encryption should be covered in this layer. This internal communication system should also work as a wrapper and a single point of entry to the NE, ensuring that the protocol is followed. Quick "work arounds" such as direct calls to the MRI should not be done, as testing, debugging and maintenance will become harder.

Security is not an issue if the program is executing in a closed network configured solely for that purpose¹⁶, but such might not always be the case. Applications might want to access an Erlang system through an open network.

¹⁶ For example, a closed network managing a telecom application.

It is very questionable if the security mechanisms that exist in distributed Erlang are acceptable to run applications on open networks. By configuring the net kernel, message passing among nodes could be limited to a registered process. This will stop all processes on other nodes from executing remote procedure calls, spawning processes, or executing any other operation allowed in a distributed Erlang environment.

Message passing, remote procedure calls, and all other inter-node communication is not encrypted. A secure socket layer library allows data exchanged through port communication to be encrypted. Using ports between Erlang nodes, however, poses limitations and hides the transparency provided by the Erlang distribution described in this section. Another option would be the crypto application, which encodes and decodes Erlang terms. Performance would in both cases be affected. Several research papers have been written suggesting how Distributed Erlang can be made more secure. For ongoing work on safe Erlang, see [Brown, 2000].

Generic Utilities

Generic tools can easily be implemented and reused across projects. These tools can be seen as agents and bridges, simplifying the mapping from standardized management protocols, providing services not handled in the MRI. They can also act as caches and proxies, handling local optimizations.

Bridging and Interfacing

The *get_next* operation is the key operation in SNMP. It was not included in the MRI as it is too closely related to SNMP, and not implemented in other standardized protocols. Implementing a generic function handling the *get_next* request on all MRI tables, however, would not only allow the re-usage of code, but it would also allow local optimizations.

In order to implement the *get_next* operation, three steps are required. The first is to call the *get_keys* function. The list it returns is scanned for a lexicographically larger key which is used in the *get_info* call for that managed object. The *get_keys* is an expensive operation, as it includes a match on a table and the displacement of a lot of memory to transfer the keys. The match operation will affect the real time properties of the system as it is implemented using a BIF, while the memory displacement might in turn trigger the garbage collector. An optimal solution would be to cache the keys, and refresh them based on some predefined algorithm. When using SNMP, there is no guarantee that the state of the managed entities displayed in the EM is up to date with the one stored in the CP, so caching would greatly improve performance with no risk for further inconsistencies.

The *get_next* operation could be generalized even more and be called *get_more*. Passing the number of MOs one wants to retrieve to the *get_more* would return a list of MRI records, allowing for the efficient implementation of the GET-BULK request in SNMP. This reasoning goes further and includes the ability to filter on certain parameters in the MRI record keys or any of its fields.

Generic Tools

Generic subsystems can be written to work with the MRI, and be reused in different applications. These subsystems could include generic software upgrade routines on a device processor level, alarm filtering and suppression, as well as performance monitoring. The MRI provided the exact division between the logical parts of the system and the physical one, greatly facilitating the generic aspects of the implementation.

One of the tools implemented in the AXM3 project was a configuration manager that created and used system configuration files based on MRI actions. By starting a central processor and a management system, an operator could use the logic of the existing software to configure the system. A subsystem in the MRI monitored all the MRI actions, and if the execution was successful, it logged it on a file. An action was successful, if all the range and syntax checks in the SNMP agent were successful alongside all the logical checks in the CP. The result was a machine made syntactically and semantically correct file, as it was the same logic that checked configurations on a live system. This file could then be read and played on a real system, configuring it upon initial starts or during runtime when extra hardware nodes are added.

Design Flaws

MRI operations

The *get_keys* function proved to work well as long as the tables had less than a couple hundred elements. As soon as the table got larger, retrieving the list of keys, sorting it and forwarding it to the process executing the request caused bursts of memory usage. In the ANx project, these problems appeared on the cross connection table, where the *get_keys* function had to be executed often in order to retrieve small subsets of keys. This problem was solved by allowing filtering on certain tables. The filtering, unfortunately, was implemented by inserting a new function in the MRI. This function was not standardized and implemented everywhere.

The syntax of the *get_keys* function had originally been defined to allow a future expansion with filtering on the keys. It takes an empty record as an argument, so by setting fields in this record, filtering functionality could have been added uniformly with little impact on the protocol. The functionality was never implemented in the first version of the MRI as its need was not certain. Simplicity was chosen over complexity, leaving the option open.

Matching on table keys is efficiently implemented in ets tables. Had that not been enough, a study on how the matching on fields other than the keys would affect the design. This was originally never an option, as it would have greatly increased the complexity of the CP code under the MRI, not allowing the contents of a MRI record to be spread across several tables. Other limitations included issues such as filtering on data retrieved from hardware in real time. It is also possible that local

optimizations could have been achieved above the MRI, solving this problem, and it is unfortunate that neither approach was taken.

Erlang

A major problem in Erlang has already been covered, namely the lack of efficient low-level ets operations allowing the retrieval of the table in chunks. In the R7 OTP release, ordered ets tables were implemented. This means that for large tables, the sort operation is not necessary any more when retrieving all the keys. In OTP R8, an additional match function returning a specified number of elements is available for every call. This would have been a perfect solution, but was only available through the use of mnesia queries and cursors, an operation which was too CPU intensive for a soft real time system. None of these options were available at the time.

A problem remaining, however, is the lack of functionality available for the generic manipulation of records. This lack of functionality makes it very hard to implement generic reusable code for record manipulation. In the AxM3 project, a completely generic MRI core was implemented using callbacks. This, unfortunately, was done using the tuple implementation of records. BIFS to generically create, update and retrieve data from records are needed to take full advantage of Erlang's dynamic type system.

A full set of required functions follow. They can be easily implemented in the preprocessor and inserted in modules using the include files where the records are defined.

- `create_record(RecordType) => #RecordType{ }`
- `set_record_field(Record, Field, FieldValue) -> NewRecord`
- `get_record_field(Record, Field) -> FieldValue`
- `record_fields(RecordType) -> [Field1, ..., FieldN]`
- `record_type(Record) -> RecordType`

Apart from these two problems, Erlang/OTP was once again proved to be an excellent tool in the development of distributed soft real time control systems.

Conclusion

This section covers the conclusions from the two implementations that have so far been made using the Managed Resource Interface.

The MRI encourages incremental development of systems. It is a natural interface that allows the subsystem to be broken down into subtasks that can be developed and tested independently before being integrated. It also allows subsystems to be added at a later stage of the development phase, allowing engineers to concentrate on core functionality above details such as graphical tools or standardized protocols. MRI also allows subsystems to be interchanged or rewritten with little effect on the overall software. Reusability is also an important element which we have been aided and made possible. All of these arguments follow the philosophy that has to be adopted when working with Erlang, and attempts to enforce it where it is not obvious.

The MRI was first used in the rewrite of the ANX. After the rewrite, it allowed a small group of software engineers to quickly and efficiently implement an HTTP based management system. The Local Craft Terminal, a tool used by operators out on the field, was implemented with no impact on the existing CP software. In addition, this team was able to test their code by reusing the stubs the engineers working on the SNMP rewrite implemented. The original aim of the MRI was thus achieved.

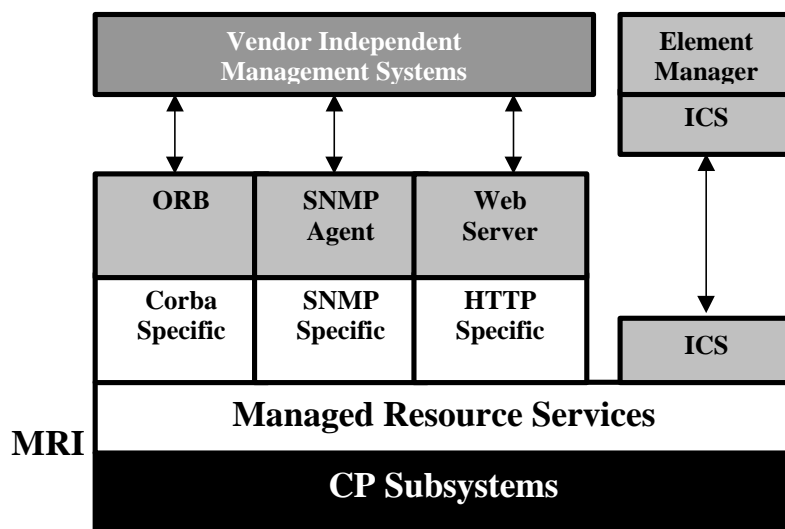


Diagram 10: *The perfect system, where different proprietary and standardized management protocols coexist in the same CP without influencing the software architecture of the underlying subsystems.*

The AxM3 project, on the other hand, was started from scratch with the MRI in its original design. The MRI had successfully been used as an interface in the prototype, giving simple tools controlling the system a protocol that was first used locally on the same node and across a network through the use of RPC:s.

The AxM3 project reused the ANx MRI Information model for all equipment and alarm handling, providing the first defacto Information model. This allowed for large parts of SNMP specific code and Managed Resource tools from the ANX to be reused, including parts of the MRI stubs.

The project, in turn, resulted in several generic subsystems in the NE interfacing the MRI. These subsystems, including equipment and alarm handling, can in turn be used in any telecom software platform. This base, along with all the experience which has been collected in this area so far, will now allow for the implementation of the perfect system. A system where standardized management protocols can be placed alongside each other with out affecting the architecture of the logical parts of the software architecture, providing the flexibility required to quickly adapt to changing market requirements or information models.

Bibliography and References

- [Armstrong et al., 1995] J. Armstrong, R. Viriding, K. Wickström, and M. Williams, Concurrent Programming in Erlang. Prentice Hall, 1995
- [Berkhout, 1994] V. Berkhout, SNMPv2 Simple or Sophisticated?, University of Twente, The Netherlands, 1994
- [Björklund and Eriksson, 1995] M. Björklund and K. Eriksson, A Framework for SNMPv2 in Erlang, Master's Thesis TRITA-NA-E9521, Royal Institute of Technology, Stockholm, Sweden, 1995
- [Brown, 2000] Dr. L. Brown, Custom Safety Policies in Safe Erlang. Proceedings from ACISP2k, Brisbane Australia. June 2000
- [Clemm, 1996] A. Clemm, Management Protocols and Their Tradeoffs. In proceedings of the Supercomm 96 conference, Dallas, TX, USA, 1996.
- [Clemm, 1997] A. Clemm, SNMP and TL-1: Simply Integrating Management of Legacy Systems, Ericsson Fiber Access Internal Report, Menlo Park, CA, USA, 1997
- [OTP 4.8] Erlang/OTP 4.8 reference manual, Ericsson Utvecklings AB, Stockholm, 1999
- [Edwards et al., 1997] J. Edwards, D. Harkey, R. Orfali, Instant Corba, Wiley Computer Publishing, 1997.
- [Hegering and Abeck, 1994], H. Hegering, S. Abeck, Integrated Network and System Management, Addison Wesley, 1994
- [ISO 7498/4] Information Processing Systems, OSI Basic Reference Model part 4: Management Framework, Geneva, 1989
- [ISO 10040] Information Processing Systems, OSI Systems Management Overview, Geneva, 1989
- [Keisu, 1998] T. Keisu, How Erlang/OTP Promotes New Design Techniques, proceedings of the ESSI 98 conference, Frankfurt, Germany, 1998.
- [Oslebo, 1998] A. Oslebo, Web-based Network Management, Dept. of Telematics, Norwegian University of Science and Technology, 1998
- [Perkins, 1993] D. Perkins, Understanding SNMP MIBs, Synoptics Communications Inc., 1993
- [Pras, 1995] A. Pras, Network Management Architectures, Ph.D thesis ISSN 1381-3617; no 95-02, University of Twente, The Netherlands, 1995

- [RFC1157] J. Case, M. Fedor, M. Schoffstall, C. Davin, Simple Network Management Protocol (SNMP), May 1990
- [RFC 1213] K. McCloghrie, M. Rose, Management Information Base for network management of TCP/IP-based Internets: MIB II, March 1991
- [Rose, 1994] M. Rose, The Simple Book, An Introduction to Management of TCP/IP-based Internets, Prentice Hall, 1994
- [WEB] IEFT RFC list, <http://www.ietf.org/rfc/>
- [X208] Specification of Abstract Syntax Notation One, CCITT recommendation X.208, Geneva, 1987

Acknowledgments

First to be mentioned is Joe Armstrong, who through patience, blood, sweat and tears taught me how to write beautiful and effective Erlang code.

A very special thank you goes to Mike Williams, who soon followed after and taught me all the secrets of large scale software design. Without him, the MRI would never have seen the light.

Leif Johansson, at Ericsson Radio for all the brain storming sessions in regards to the development of generic reusable tools, and on his feedback on SNMP.

Chandru Mullaparthi at T-Mobile UK deserves a special mention for his knowledge and feedback on standardized management protocols. If he had an ISBN number, you would have found him in the reference section of this thesis.

Denise Stack, at Ericsson Expertise Ireland, for her valuable feedback on the first versions of this thesis and for all the brain storming sessions during the second implementation of the MRI.

My mother, for comments on grammar and spelling. She now knows more on the MRI that I do.

And every one else... You know who you are! As you are many, I hope a collective thank you will do. You have helped in the research, in explaining the unexplainable, and read this thesis and providing valuable feedback. Thank you!

Appendix A: Erlang

This chapter gives the necessary background on Erlang, the programming language used to develop the Managed Resource Interface described in this paper. It goes into detail on the aspects of the language needed to understand the managed resource interface.

Language Abstract

Erlang is a symbolic high-level functional programming language. It was originally developed for programming and prototyping telecommunication systems, but due to its properties quickly proved to be ideal for other industrial applications dependant on features such as fault tolerance, concurrency, distribution, and soft real time properties.

Programs in Erlang consist of functions stored in modules. Functions can execute concurrently in lightweight processes, and communicate with each other through asynchronous message passing. The creation and deletion of processes require little memory and computation time.

In Erlang, memory is dynamically allocated. A real time garbage collector deallocates memory when it is no longer referenced to. Shared memory does not exist, forcing processes to exchange data through message passing.

An error handling mechanism allows a process to trap exceptions created from incorrect expression evaluations, attempts to execute undefined functions, and failing match operations. Exceptions can also be generated by the user changing the program's flow of control. Processes can monitor each other through links, and either be deleted or receive notifications depending on their execution state, and that of the processes they are linked to.

An important property of the language is its distribution mechanism based on TCP/IP. Processes can interact with each other on different physical machines, or on the same machine running different Erlang Systems. The syntax for the interaction is the same as the system used for local process communication, allowing programs written for a single environment to be easily ported to a multi-processor system. The distribution mechanism ensures transparency between the environments, even when the machines are running different operating systems.

The Erlang system offers a set of built in functions called BIFs. They are used to execute tasks not expressible with Erlang's primitives. They include data type conversions, distribution mechanisms, system information retrieval, and operations dealing with ports and processes.

In the development environment, functions are evaluated using the shell. It is a simple interactive command interpreter allowing the user to load, compile and evaluate expressions. It also allows the user to inspect and control the state of the system.

Final programs are executed without the shell. They are started up through scripts where start functions and necessary parameters are specified. The target environment can be anything from a PC to an embedded system.

Data Objects

Variables in Erlang are single assignment. They are instantiated when created, and may not be assigned new values. Variables are dynamically typed, and need not be declared. Atomic storables are integers, floats, atoms, refs¹⁷ and pids¹⁸. Four compound data types are provided. Tuples, records¹⁹ and binaries are of fixed size, while lists are of variable size. Macros can also be defined.

Data type	Expression	Value of B
Tuples	A = {1, two, 3}, B = {A, 4, 5}.	{{1, two, 3},4, 5}
Lists	A = [two, 3, 4, 5], B = [1 A].	[1, two, 3, 4, 5]

Example 1: *The result of nesting a tuple and appending an element in a list, storing the result in B*

Records are a key concept in the MRI. Together with actions, are used to specify the system specific parts of the interface²⁰. When records are defined, their name and fields are specified. A definition can appear anywhere in the code, as long as it is before a clause using the record. It is good practice to put record definitions in separate files included by the modules using them. A default value can be assigned in the field declaration. If no value is specified, the field is automatically assigned the atom undefined. Records are bound to variables, and when a field in a record is changed, it has to be bound to a new variable. Guards exist on records, allowing checks to see if a record is of a specific type.

Operation	Syntax	Example
define	-record(Name, {Field1 = Val1, Field2, ..}).	-record(person, {age, name = "Dee Dee"}).
create	Var = #Name{Field1 = Val, ..}.	Person = #person{age = 12}
update	NewVar = Var#Name{Field = NewVal, ..}.	NewPerson = Person#person{age = 13}.
access	Var = Record#Name.Field	Name = Person#person.name
guard	When record(Name, Record) -> ...	when record(person, Person) -> ...

Example 2: *Operations for defining, creating, and manipulating records.*

¹⁷ A unique reference for the Erlang system.

¹⁸ A process identifier.

¹⁹ Records are translated by the pre-processor to tuples, and operations on them to functions applied to these tuples.

²⁰ The information model.

Functions and Modules

A system implemented in Erlang is divided up into modules and include files. Functions declared in the modules are local to the specific module if not explicitly exported. Exported functions can be evaluated by functions in other modules or through the shell.

```
-module(math).  
  
-export(factorial/1).  
  
factorial(0) -> 1;  
factorial(N) -> N * factorial(N - 1).
```

Example 3: *A module called math, where the exported function factorial(Integer) can be used by other modules, evaluated by external processes, or called through the shell.*

Pattern Matching

Pattern matching is a key concept in Erlang. It is used for variable assignment and clause choices, altering the flow of the program. Patterns have the same structures as Erlang data objects, but can include unbound variables that are bound when the matching occurs.

A function with the same name can be defined several times, where through pattern matching, the clause chosen depends on the parameters and arguments passed to it. Examining example 3, if the argument in the function call is not 0, the second clause is automatically chosen, and the argument is bound to N. If the argument is 0, the first clause is chosen, and 1 is returned.

Flow of control

There are three control structures in Erlang, the case statement, the if statement, and the receive statement. Pattern matching is used to determine the flow of control, and to assign variables in these structures. Guards can be used to define conditions that have to be fulfilled in order for the clause to be chosen. They are optional, and may not be defined by the user.

```

factorial(Int) ->
  case Int of
    Neg when Neg < 0 -> {error, negative}
    Int when integer(Int) -> math:factorial(Int);
    _ -> {error, typeMismatch}
  end.

```

Example 4: A case statement that checks that the arguments to factorial are positive integers, returning either the factorial of the integer, or the tuple {error, Reason}.

```

If
  Guard1 -> Sequence1;
  Guard2 -> Sequence2;
  .....
  GuardN -> SequenceN;
  true -> SequenceN+1
end

```

Example 5: The syntax of the if statement.

In if and case statements, should no match exist, the process will terminate with an error. In case statements, the user can avail himself of the special variable ‘_’, used as a placeholder for cases where we are not interested in the value. This match will always succeed. In if statements, the atom true can be used instead of a guard, executing that sequence should none of the guards succeed.

Concurrency

Erlang processes are created dynamically by evaluating the spawn/3²¹ BIF. The spawn BIF takes a module name, a function name, and a list of parameters for the function as arguments. This is the first function that is called by the new process. The BIF call returns a PID, which is used to identify this process.

Messages are sent to the process through the *Pid ! Message* construct. A message can be any Erlang expression or term. They are received as terms in a receive clause, and bound through pattern matching. If messages cannot be matched, they remain in the processes’ mailboxes until the flow leads into another receive construct where an appropriate match occurs. Time-outs are allowed in receive statements, should no message match or be received.

²¹ The /3 in the spawn BIF denotes that the call takes three arguments.

```
Receive
  Message1 -> Sequence1;
  Message2 -> Sequence2;
  .....
  MessageN -> SequenceN;
  after Time -> SequenceN+1
end
```

Example 6: *The syntax of the receive clause.*

In order to facilitate message passing, processes are allowed to have static identifiers represented as atoms. These identifiers are assigned through the `register/2` BIF, allowing this identifier to be used instead of the `Pid`. One process at the most may be registered with a specific identifier at any one time in an Erlang system.

```
Pid = spawn(server, start, [initial]),
register(server, Pid),
server ! hello,
Pid ! bye, ....
```

Example 7: *Code that spawns a process, registers it, and sends two messages to it.*

In Example 7, a process executing the function `start/1`, located in the module `server`, is created. The atom `initial` is passed as a parameter. This process is then assigned the name `server`. Two messages are sent to it, the atoms `hello` and `bye`. `hello` is sent through the use of the process name assigned to it, while `bye` is sent to it through the use of the process identifier.

Distribution

An Erlang node can be thought of as a complete Erlang system. We call a network of two or more Erlang nodes interacting with each other a distributed system. The nodes are identified by a name given to the node during the start-up of the Erlang system, and by the name of the machine they are executing on. The BIF `node/0` returns the name of the node the expression is evaluated in. Assume the node with the name `server` is running on the machine `ramones.ericsson.se`, the name of the node would be `server@ramones.ericsson.se`²².

The primitives for communication between processes in different nodes are the same if `Pids` are used. Should the user want to send a message to a registered process on another node, however, the node name must be specified.

²² Or simply [server@ramones](#) if we are operating within the same domain and use short names.

Node names are used when operations from one node to another include spawning processes, executing remote procedure calls, or monitoring of these remote nodes.

```
Pid = spawn(server@ramones.ericsson.se, server, start, [initial]),  
Pid ! bye, ....
```

Example 8: *The same as in example 7, only that the process is spawned in another Erlang system on the machine ramones.*

The programmer does not need to set up connections between nodes. They are set up by the run-time system when a node, for the first time, is referred to by another node. When the connection has successfully been set up, the newly connected node adds its peer to the list of known nodes, spreading the information to other nodes they were previously connected to. This creates a loosely coupled system, where node configuration can change dynamically, whenever nodes are added or removed.

Authentication

The creation of loosely coupled systems involves some security aspects that are only partially handled by the underlying system. Every node has a secret cookie assigned to it. If a set of nodes shares the same cookie, they can freely communicate with each other, and no restrictions are imposed as to what operations can be performed on the individual nodes by processes in other nodes.

Open Telecom Platform

The Erlang system has a set of libraries that provide building primitives for larger systems. They include routines for I/O, file management, and list handling. Other libraries handle lists of ASCII²³, parsing, generic servers, etc. The GS library provides a set of graphical event driven routines, while the ets²⁴ library provides a hash table implementation for accessing data in constant time. This set of standard libraries is seen as an application in the Open Telecom Platform.

Large systems written in Erlang use the Open Telecom Platform. It consists of a set of design principles, which together with middleware applications yields building blocks for scalable robust real time systems. Supervision, restart, and configuration mechanisms are provided for included applications, and specified for user-defined applications allowing a standardization of behaviors between systems.

Provided applications are constantly expanding and today include SNMP agents, complete with MIB compilers and guidelines for instrumentation functions. A fault

²³ Strings do not exist in Erlang, and are instead represented as lists of ASCII characters.

²⁴ Erlang term storage

tolerant HTTP server facilitates the development and integration of web-based interfaces. A distributed relational database called Mnesia, with its own query language mnemosyne, allows transaction handling in a distributed environment. An Orb facilitates the development of CORBA based management systems. Interfaces towards other languages include Jive, a Java interface, IG, an interface allowing Erlang programs to call c modules, and c modules to call Erlang. These interfaces are complemented with the possibility of defining IDL interfaces, through which code can be generated. Other included applications involve deal with error handling, alarm handling, monitoring, and more²⁵.

Design Techniques and Methodology

Like any tool used to solve a problem, there are correct and incorrect ways of going around in problem solving. Using a methodology proven successful in other types of development is no guarantee that it will work when developing systems in Erlang/OTP.

Erlang's properties as a high level language introduces a simple edit compile and test cycle. Prototypes can thus be quickly and efficiently implemented, testing ideas at an early stage of development. Features such as dynamic typing, flat module, process concepts and application structures allows for an incremental development of the system. Functionality can be added either as components on already existing modules, or as layers on the software. This facilitates testing in between the increments, and allows errors and inconsistencies to be detected at early stages of the development.

As Erlang is a higher-level language, there is a no need to document as many levels between the requirements and the implementation. Documents necessary are subsystem descriptions, with their respective interfaces. Interfaces should remain stable between the increments. This has proven to be possible due to the polymorphic properties of the language. An application description describes how the subsystems interact between each other, and a high-level system-architecture document describes how the applications interact among each other on one processor.

For more in-depth reading on how Erlang / OTP promotes new design techniques, see [Keisu, 1998].

²⁵ For an up to date list, see the OTP reference manual and user guide.

Appendix B: MRI Example

The following chapter describes parts of the equipment and alarm models from the ANx project, a telecom application providing households with broadband access. It gives a practical example of what has so far been covered in the abstract description of the functional and informational models of the MRI.

The Equipment Model

The equipment model described is hierarchically organized. On the top level, there is a system node. It logically groups together equipment, either geographically, functionally, or both. The subrack is the hardware in which the boards are inserted. Every subrack is associated to a system node, so both values have to be included in the record to yield a unique key.

```
-record(mr_sn, {sn = key,      % int(), Node identifier
               mgt}).      % Management Data

-record(mr_sr, {sn = key,      % int(), Logical subrack grouping
               sr = key,      % int(), Subrack Id
               hw_id,        % str(), Hardware Id
               type,         % adsl, maccg, other
               alarm_status, % [], Subrack Alarm Status. See MRI
               mgt}).      % Management Data
```

The keys of these record are the sn and sr fields. They are always given the default value key when the record is defined, and always have to be set to an actual value whenever a get_info call is executed. A get_info call to retrieve the MRI record containing the information for the system node 1 and the subrack 10 would be

```
mr:get_info(#mr_sr{sn = 1, sr = 10}).
```

The return value of this call would be either {error, Reason} or {ok, Record}. The error reason is defined in the functional model to be the atom type or instance. Record is an instance of the #mr_sr record, where all the fields are set.

Slots are positions in the subrack where the boards can be inserted. Slots are never created or deleted individually, as they are directly associated with the subrack. When a subrack is created, all the entries in the slots table are also created. The same applies to deleting a subrack. All instances of the slots in that subrack will also be deleted.

```
-record(mr_slot, { sn = key,      %int()
                  sr = key,      %int()
                  spos = key,    %int(), Denotes the slot position
                  type,         %virtual | real
                  configured,    % unmanaged | unconfig_empty | config_empty | config_board
                  alarm_status,  % [removed | unconfig_inserted | mismatch | unrecognized]
                  mgt}).
```

The board records denote the managed entities that are inserted in the slots. There is a one to one relation between a board and a slot, so the keys are the same. Many different types of boards can be inserted in the slots, and parameters from these different boards may differ. The board record contains all the values that all boards can have in common. Other values are placed in record extensions, resulting in records that share the same keys as their abstract counterpart.

```
-record(mr_board, { sn = key,      % int()
                  sr = key,      % int()
                  spos = key,    % int()
                  type,          % et | power | qau | ce | ucm | hfc_nt | adsl_nt
                  hw_id,         % string()
                  sw_id,         %{Core, Appl} string()
                  status,        %{Oper, Admin} up | down
                  led_status,     % red_on | red_off | none
                  alarm_status,   % see alarm list in MRI document
                  mgt}).

-record(mr_hfc_slot, { sn = key,    % int()
                     sr = key,     % int()
                     spos = key,   % int()
                     coax,         %Logical Unit ID
                     qam,          %{Spos, Index} Upstream Port
                     ucm,          % {Spos, Index} Downstream Port
                     mgt}).
```

The `mr_hfc_slot` record will only exist if a slot is configured to contain a board of type `hfc_nt`. It will contain a logical identifier for the coaxial cable it is connected to, together with the ports that handle the upstream and downstream traffic when the board is inserted. This information is specific only to slots that will contain an hfc Network terminal.

The hierarchy in the ANx goes further to include interfaces, which denote points used for inter - board communication. These interfaces are in turn divided up into entities such as points and timeslots. Just like the records in this example, interfaces and other hierarchically lower managed entities have the keys of the entity on a higher level, with an additional key.

The call to retrieve all the keys takes only one argument, namely an empty MRI record. Only the record type is checked.

```
mr:get_keys(#mr_sr{}).
```

The return value is `{error, type}` should the record not be supported. The tuple `{ok, Keys}` is otherwise returned. `Keys` denotes either an empty list, should no subtracks exist, or a list of lists, where every sub list contains the system node and the subtrack identifier, in the same order as they were defined in the MRI record. They keys are lexicographically ordered, and could for example be a list resembling `[[1,1], [1,2], [1,10], [2,5], [2,6]]`.

The following action signs on²⁶ a network terminal, (NT²⁷) located in the virtual slot found in subrack 1 at position 1, belonging to the logical grouping represented by system node 1. Rate is the bandwidth assigned to the NT.

```
mr:action(sign_on, #mr_adsl_slot{sn = 1, sr = 1, spos = 1, rate = {Up, Down}}).
```

The return value is either {error, Reason}, where Reason is either *instance*, should such a slot not exist, {*invalid*, *rate*}, should the bandwidth values entered by the operator be wrong, or *bandwidth*, should the network not be able to reserve enough bandwidth for the NT. If the operation succeeds, {ok, Rec#mr_adsl_slot} is returned. All the fields in the record are set.

The fact that this action returns a record where the values of the field refer to the valid parameters after the success of the action might lead to the thought that all actions are synchronous. This is not necessarily the case, and if the call is synchronous or not is decided in the application to which the action is forwarded to.

An example of an asynchronous call requiring an extra state parameter is the signing on of the adsl slot. Such an operation can take a long time considering that the board has to be located in the network. The signing on of a slot reserves the bandwidth and initiates the search for the board. It could take days before the board is actually connected to the network so the response time would thus not be acceptable for the operator if the calling process is to hang until the board is located. Some sort of feedback is thus given as soon as the action has been initiated. The additional state *signing_on* is used as a complement to the *signed_on* and *signed_off* states in order to return an accurate *mr_adsl_slot* record through the MRI. As soon as the signing on succeeds, an event is generated.

²⁶ Locating and activating.

²⁷ The endpoint of a broadband network into which phones, TVs, Ethernet links and other devices can be connected to.

The event and alarm model.

Here follows the MRI alarm model used for the ANx. The alarm model is the smallest of the models, and easiest to understand as it contains no project specific data other than the network element specific alarms specified in a configuration file. This makes the alarm information model product independent, allowing it (and the code written to implement it) to be reused in different products. All actions in the alarm model are also included.

The alarm model is responsible in reporting events and alarms. Events and alarms are logged in the `mr_event_log` table, and active alarms are stored in the `mr_alarm` table. These records are identical, in that they contain an index to uniquely identify the alarm or event, and an `mr_notification` record, which stores all the necessary data. Treatments on the various events and alarms are stored in the `mr_treatment` table. There exists one entry in the table for every type of alarm or event, and determines if the alarm is to be logged or sent.

The `mr_notification` record has no keys. It is just a record extension collecting data related to a specific alarm or event, and stored in one of the fields of the `mr_event_log` or `mr_alarm` records. This is possible because alarms and events consist of the same information.

```
-record(mr_notification,{ name,          % atom(), alarm or other event
                        originator,    % MRI record,
                        severity,      % minor | major | critical | warning | undefined
                        time,          % {{Y, M, D},{H, M, S},{ '-'| '+' , H, M }}
                        info,          % string(), < 255 chars
                        mgt}).

-record(mr_treatment,{ name = key,     %atom(), alarm or event see MRI
                      type,          % alarm | state_change | other_event
                      logging,        % enabled | disabled
                      sending,        % enabled | disabled
                      last_time_sent, % {{Y, M, D},{H, M, S},{ '-'| '+' , H, M }}
                      severity,       % minor | major | critical | warning |
                                      % undefined
                      category,        % undefined | communication | qos | procesing
                                      %equipment | environmental
                      mgt}).

-record(mr_event_log,{ index = key,    %integer()
                      record,         % #mr_notification record
                      mgt}).

-record(mr_alarm,{ index = key,        %integer()
                  record,
                  mgt}).
```

The configuration file contains all of the data that is inserted in the event treatment table upon start up. This file will change with the information model, and is proprietary to the CP. The name, type, severity, category and originator are however defined in an alarm appendix of the MRI document. All values are static except for the severity, which can be changed by the operator. The value described in the MRI

denotes just the default value. An alarm table describing all possible events and alarms follows.

Name	Type	Category	Severity	Originator
communication_lost	alarm	communications	major	#mr_sr
fan_failure	alarm	equipment	major	#mr_sr
power_failure	alarm	equipment	critical	#mr_sr
door_open	alarm	equipment	minor	#mr_sr
subrack_mismatch	alarm	equipment	minor	#mr_sr
subrack_created	other_event	undefined	undefined	#mr_sr
subrack_deleted	other_event	undefined	undefined	#mr_sr

There are four actions that can be applied on the MRI alarm model records. `set_event_treatment` sets the treatment for one event. Fields considered are logging, sending and severity. If any of them has the value `undefined`, it is ignored. All other values are set. An instance error is returned if the event or alarm that we want to change is not represented in the record.

```
mr:action(set_event_treatment, Rec#mr_event_treatment) -> {ok, NewRec#mr_event_treatment} |
{error, instance}
```

`Set` and `get` log properties is applied to the event log table. There is however no record denoting the event log, so the managed entity field is kept empty, and the properties are instead passed in the field used for the extra arguments not included in the managed entity record. It is noted in the next chapter that a log properties record should in fact have been defined. Setting the log properties includes changing the administrative status to `enabled` or `disabled`²⁸. `Size` regulates the number of records stored in the table according to the first in, first out principle. This action can be used to clear the log by setting its value to zero, and then back to its original one.

```
mr:action(set_log_properties, {}, Properties) -> {ok, [{admin_status, up | down}, {size, Size}]}
```

```
mr:action(get_log_properties, {}) -> {ok, [{admin_status, up | down}, {size, Size}]}
```

```
mr:action(get_no_of_alarms, {}) -> {ok, NoOfCurrentAlarms}
```

²⁸ Log or don't log all `mr_event_treatment` records whose log treatment is set to `enabled` | `disabled`.

Appendix C: Abbreviations

ADSL	Asymmetric Digital Subscriber Line
API	Application Programming Interface
ASN.1	Abstract Syntax Notation 1
ATM	Asynchronous Transfer Mode
BIF	Built In Function
CDMA	Code Division Multiple Access
CMIP	Common Management Information Protocol
CMIS	Connection Management Information Service
CORBA	Common Object Request Broker Architecture
DMTF	Desktop Management Task Force
ETSI	European Telecommunications Standards Institute
GPRS	General Packet Radio Service
GSM	Global System for Mobile Communications
HTML	Hyper Text Mark-up Language
HTTP	Hyper Text Transfer Protocol
IAB	Internet Activities Board
IDL	Interface Definition Language
IEEE	Institute of Electrical and Electronics Engineers
ICS	Internal Communication System
IDL	Interface Definition Language
IETF	Internet Engineering Task Force
IIOF	Internet Inter-Orb Protocol
ISDN	Integrated Services Digital Network
ISO	International Standardization Organization
IP	Internet Protocol
ITU-T	International Communication Union
JMAPI	Java Management API
NE	Network Element
MIB	Management Information Base
MO	Managed Object
MRI	Managed Resource Interface
OID	Object Identifier
OMG	Object Management Group
ORB	Object Request Broker
OSI	Open System Interconnection
OTP	Open Telecom Platform
PDU	Protocol Data Units
PSTN	Public Switched Telephone Network
RFC	Request For Comments
SGML	Standardized General Mark-up Language
SMO	Systems Management Overview
SNMP	Simple Network Management Protocol
TCP	Transmission Control Protocol
TMN	Telecommunication Management Network
UDP	User Datagram Protocol
WBEM	Web Based Enterprise Management