THE DESIGN OF A FLEXIBLY INTERWORKING DISTRIBUTED MESSAGE-BASED FRAMEWORK

Bilhanan Silverajan, Ilkka Karvinen, Janne Mäkihonka, Jarmo Harju
Department of Information Technology,
Tampere University of Technology
P.O. Box 553, FIN-33101, Tampere, Finland
e-mail: {bilhanan | ik | mhonka | harju}@cs.tut.fi

ABSTRACT

The DOORS (Distributed Object OpeRationS) framework attempts to address several issues in protocol development, interoperability and distributed computing for client-server architectures in heterogenous networks and telecommunications systems.
Apart from supporting TCP/IP and UDP/IP socket-based communication, the DOORS framework offers an additional level of interworking with many kinds of distributed object models and distributed systems. It also implements an extremely lightweight ORB using dynamic interfaces running over IIOP which conforms to the CORBA architecture. Interworking with other architectures and frameworks is supported via interrupt-driven, CORBA, or protocol-level approaches.
Brief examples of these approaches, together with a description of streaming audio applications developed for use with this framework, are presented.

1. INTRODUCTION

Telecommunications networks and devices, as well as the Internet are inherently comprised of extensively distributed network and software-based elements and equipment. Owing to recent emerging trends of the convergence of communications and computing, architectures and infrastructures in the telecommunications domain have become significantly dependent on advances in areas of distributed computing such as CORBA[1], mobile agents, object-oriented languages and reusable design techniques such as design patterns for improving quality and scalability. One exciting topic in the area of distributed systems is event-based systems development in which the need occurs to support emerging applications that require dynamic responses to asynchronous distributed occurrences.
With the adoption of such new technologies, however, one cannot ignore the issue of large amounts of legacy infrastructure that already exist, and their integration or interworking with newer technologies with minimum adverse effects.
Using Ad-Hoc methods such as wrappers to encapsulate entire systems or applications, tends to be complex, fragile, and error-prone, often introducing instability into a previously stable legacy application. Therefore such methods can become maintenance burdens themselves.
Thus any framework which attempts to address the problem of interworking with legacy systems while providing enough functionality to develop new distributed event-based applications needs to fulfill the following criteria:

- Implement new object interfaces that conform to a set of widely supported distributed object specifications as much as possible
- Accomplish data translation and information processing on potentially different levels for several uses such as gateways or relays
- Provide connection protocol management on different levels
- Manage the application environment in a lightweight manner.

In this paper, we present a practical high-speed framework in active development that fulfills these criteria, codenamed DOORS. The main purpose of DOORS is to overcome the abovementioned difficulties to flexibly support the implementation of

various types of distributed message-based systems and services. DOORS applications can also noninvasively interwork with existing legacy applications as well as applications and components that use CORBA or some other distributed model.
The design of this framework has been heavily influenced by existing object paradigms and design patterns, which enables it to successfully handle occurrences of discrete events asynchronously with great ease. Section 2 of this paper describes the structure of this framework and its associated libraries, while section 3 explores several interworking scenarios. Section 4 presents a short description of test applications developed for use with this framework, while section 5 discusses some of the design decisions made.

2. DOORS DISTRIBUTED FRAMEWORK

The DOORS (Distributed Object OpeRationS) framework attempts to address several issues in protocol development, interoperability and distributed computing for client-server architectures in heterogenous networks and telecommunications systems.
DOORS uses a pure message-based model and derives its C++ code-base from a protocol implementation framework, OVOPS, that has been successfully tested and employed to develop medium to complex protocols widely used in telecommunications and the Internet [2].
The framework has been implemented as three flexibly interacting library-level components, which are the Base, Protocol and Dynamic CORBA components repectively. It is presently structured in such a way as to allow multiple builds of different platforms and compilers to proceed concurrently. The current development platforms being used for DOORS are Solaris and Linux, using Sun C++ as well as GNU C++ compilers.
DOORS reduces any event-based application or protocol into a set of interacting event- and protocol tasks and a round-robin scheduler which allocates execution turns to the tasks. An entire DOORS system, comprising its event- and protocol-tasks, scheduling, I/O Handling and CORBA subsystem, is able to execute as a single user or kernel UNIX process.
The driving design principle which has been adhered to throughout the implementation of these library components is to ensure that only a modular dependency is enforced among these components. The Base Component Library provides all the services needed by the Protocol Component Library, and likewise, the Dynamic CORBA Component Library is dependent on both the other two Component Libraries.
Because all these library components are usually compiled as separate runtime libraries, a developer can flexibly customize the size of his or her event-based application very easily, with the bare minimum being an application-level task, a scheduler, an I/O Handler and possibly one or more virtual devices for inter-process communication, services which are all directly provided by the classes in the Base Component library without the need for the other two libraries. Subsections 2.1, 2.2 and 2.3 describe these libraries in greater detail.

2.1. Base Component Library

The Base Component is subdivided into three modules: Utilities, Core and Protocol Support. These interwork together by providing the following functionalities and services:

- Frames, buffers and cells for data storage
- Loggers
- Event-, Protocol- and timer management task classes
- Classes that implement Service Access Points (SAPs) and Protocol Data Unit (PDU) encoding and decoding
- Base classes for finite state machine implementations
- Multiplexers
- Base classes for messages
- Bi-directional ports used by tasks for message passing
- Inter-task Scheduler
- Virtual device classes that provide object-oriented interfaces to system devices and other operating system services, such as sockets, pipes, file handlers and timers
- I/O Handler that manages the virtual devices.

DOORS perceives any event-based application or protocol as a set of interacting event- and protocol tasks which exchange messages through their ports. Events in the system are represented as messages. Ports conceptually form channels for asynchronous message transfer via a protocol-based communication with other tasks. For endpoints that connect to an external system, virtual device classes implemented as wrappers for UNIX pipes, TCP/IP and UDP/IP sockets can also be utilised.
Tasks, as shown in a simplified form in Figure 1, execute incoming messages based on a FIFO order, requesting execution turns from a round-robin scheduler. Execution turns are non-preemptive, and if there are no outstanding requests for task executions, the scheduler instead blocks on the I/O Handler until outside events monitored by the I/O Handler and the

virtual devices it manages arrive.
Figure 2 illustrates the basic procedures involved with message-based communication between two tasks where, on obtaining an execution turn from the scheduler, Task A removes the first message from its message queue and depending on its current state, executes a sequence of operations that involve the creation of a message for Task B. Within the same execution turn of Task A, this message is then transferred to the message queue of Task B via the ports, which triggers Task B's request for an execution turn from the scheduler.

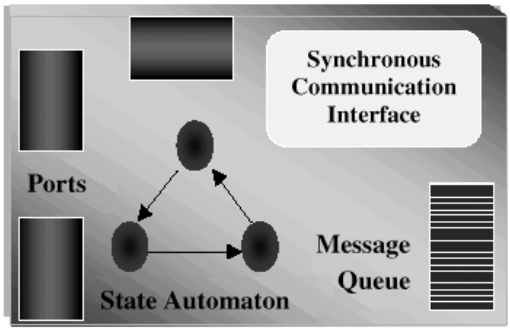

Figure 1. Task Abstraction in DOORS

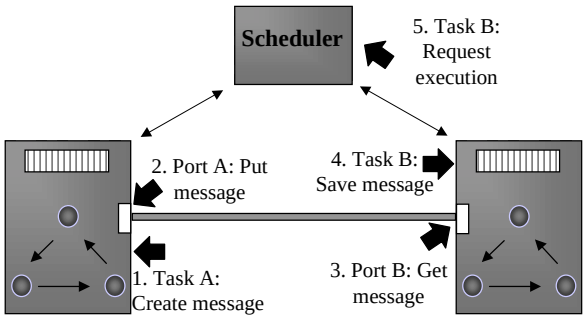

Figure 2. Message-based inter-task communication

Both the scheduler and the I/O Handler are implemented as Singletons [3], and are launched automatically at system startup.

2.2. Protocol Component Library

The Protocol component library currently contains modules which comprise protocols such as RTP (Real-time Transport Protocol), RTCP (RTP Control Protocol) [4] and IIOP (Internet Inter-ORB Protocol), which is a specialisation of the GIOP (Generic Inter-ORB Protocol) [1].
The RTP and RTCP protocols have been designed so as to support the types of unicast and multicast audio applications explained in section 4. The IIOP protocol has been designed to be utilised by applications employing the use of the lightweight CORBA services offered by the Dynamic CORBA Component Library discussed in section 2.3. Also, because DOORS is source-code compatible with OVOPS, existing production-level protocols developed with OVOPS such as TCAP, HTTP and TCP/IP can be used natively as well.
All the classes used to build the protocols within this component library are supplied by DOORS. Implementation of protocols with stateful behaviour are supported using the concept of tasks being either Entity or Connection Objects. Each instance of an Entity Object may be seen to represent one protocol (sub-)layer, and is responsible for communication with the upper and lower protocol layers within the stack. It also creates, manages and destroys tasks which are Connection Object instances, representing single peer-to-peer connections for that protocol. Figure 3 shows the implementation architecture of IIOP.
The basic principle of creating Connection Objects by the Entity Object is supported by the Abstract Factory pattern [3], which inevitably has to be extended to encompass the multiplexing, connection-handling and subsequent removal of created objects.

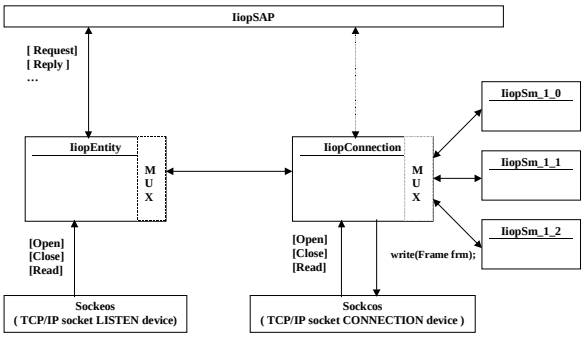

Figure 3. IIOP Implementation Architecture

In implementing support for protocol state behaviour, the base State Machine class in DOORS adopts a table-based alternative which maps inputs to state transitions using function pointers. This approach lent itself more easily to allow for the possibility of generating C++ code automatically from a set of specifications of a state machine for a given protocol using OVOPS code generators. This can then be used together with the code written by the developer. This technique was successfully used to develop the RTP and RTCP protocol modules. However, the State pattern [3] approach can also be used in DOORS, as in the case of IIOP, where there is a need for the protocol to be backwards compatible with two earlier versions. As Figure 3 shows, the protocol must be intelligent enough to allow for multiple simultaneous versions to be used, depending on the version information carried within the header field of the

protocol message.
Because there are eight different messages in the protocol and message structures differ in different versions of the protocol, the complexity of the implementation can increase rapidly if the table-driven alternative is used. Each version of the protocol also introduces significant changes from the preceding version. For example, version 1.1 introduces message fragmentation, while version 1.2 allows for clients and servers to arbitrarily reverse their roles within the same connection.
In this implementation therefore, the State pattern design approach was preferred as it allows the architecture to be easily managed and maintained without letting updates and subsequent versions affect existing versions. Figure 4 shows how this was accomplished.

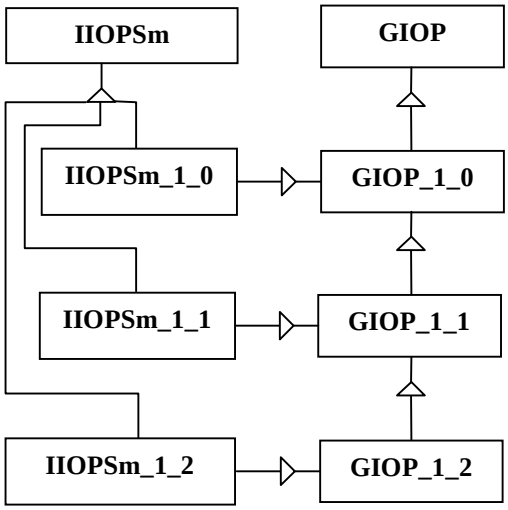

Figure 4. Finite State Machine Classes in IIOP

2.3. Dynamic CORBA Component Library

In addition to traditional IPC mechanisms using pipes and sockets, CORBA has rapidly emerged as a very realistic, mature and useful distribution technology for the Internet. CORBA has also been adopted in telecommunications by consortiums such as TINA-C [5] as the basis for its Distributed Processing Environment. Amongst all the distributed architectures in use today, CORBA is perhaps also the most realistically implementable for large-scale systems.
However a full CORBA implementation is still not a realistic lightweight possibility for event-based communications and frameworks which need to remain interoperable with legacy systems. The core specifications remain oriented towards synchronous communications, and instead introduce the overhead of a dedicated CORBA Messaging Service [6] for asynchronous communications.
Owing to these reasons, DOORS supplies a component library implementing a lightweight, library-based CORBA solution for asynchronous communications which consists of an Object Request Broker (ORB), a Portable Object Adapter (POA) and dynamic interfaces for client-side and server side in the forms of the Dynamic Invocation Interface (DII) and the Dynamic Skeletal Interface (DSI), which run atop IIOP.
The ORB is implemented as a C++ Singleton class which is derived from an Event Task, and its design corresponds to the Broker architectural pattern [7] documented by Buschmann et al. The use of the dynamic client and server side interfaces imply that the traditional stub and skeletons which usually manifest as proxy objects in the pattern are no longer present. Instead, using the DII and DSI, the clients and servants interact asynchronously with the ORB by using Request and ServerRequest objects that contain parameters reflecting the operation name and the parameter values and return results, as defined in the CORBA specifications.
Figure 5 illustrates the client-side operations for an asynchronous method invocation using DII.

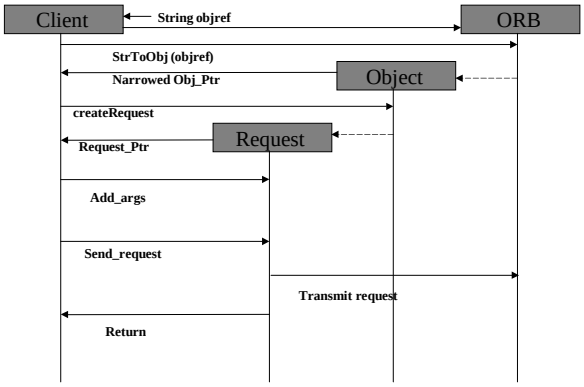

Figure 5. Client to ORB DII Communication

As the Dynamic CORBA Component is implemented using the services and classes provided by the Base and Protocol components, the same guarantee of delivery of messages which is provided by the framework for traditional protocol-based communications can likewise also be given to asynchronous inter-ORB communication in CORBA for distributed client-server applications.
Figure 6 illustrates the DSI technique in DOORS in the form of a message sequence chart, involving the ORB, POA, ServerRequest and Servant objects. The basic idea of the DSI is to implement all requests on a particular object by the invocation of the same upcall routine, a Dynamic Implementation Routine (DIR) on the Servant object. The DIR is passed all the explicit operation parameters, and it can access and operate

on individual parameters. A single DIR could also be used as the implementation for many objects with different interfaces [1].

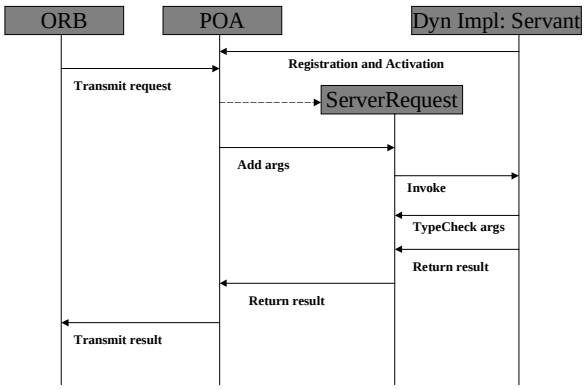

Figure 6. Servant to ORB DSI Communication

The POA that is implemented in DOORS as a Singleton relieves the ORB of the responsibility of maintaining state and persistence information about servants. It is derived from the Protocol Task and State Machine base class and maintains an Active Object Map for keeping track of active CORBA Objects and their corresponding servants. It also supports the following policies:

- Single-threaded
- System-generated Unique Object ID
- Transient server objects
- Support for server retention in the Active Object Map
- Explicit Object Activation.

3. INTEROPERABILITY WITH OTHER SYSTEMS AND ARCHITECTURES

DOORS currently supports interworking and interoperability with other architectures and systems via protocol-level, CORBA and interrupt-driven approaches. Figure 7 concisely portrays the mechanisms that can be used to achieve such interoperability.

As alluded to earlier, the entire system can, and is in fact lightweight enough to be run as a single process. It can concurrently provide multiple points of access for multi-directional traffic between the various protocols and applications in DOORS with the external world.

The protocol-level approach is perhaps the most widely applied means of communication in the Internet today, where the objects within the frameworks are able to communicate with applications of the foreign legacy system by using some form of specified application-level communication protocol over TCP/IP or UDP/IP sockets. It could be visualised as a distributed, extreme variation of the Adapter pattern [3], where the interface of an object needs to be adapted before successful communication could take place.

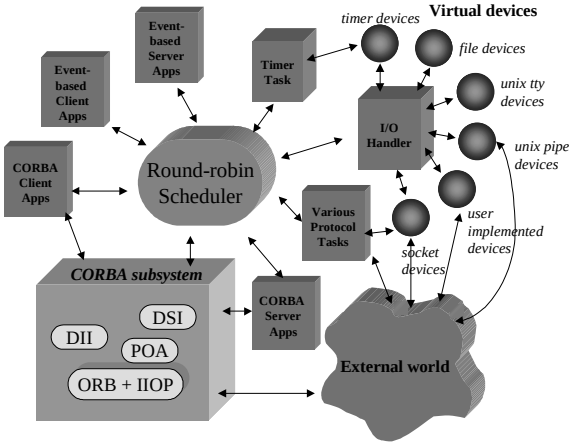

Figure 7. Interoperability mechanisms in DOORS

The CORBA-based approach allows external CORBA-compliant applications to communicate with clients and servants within DOORS via its CORBA subsystem. Although DOORS currently only implements the dynamic interfaces, this is completely transparent to the external application, which can use static interfaces with type-specific stubs or skeletons on its part. As the DII and DSI also allow type-checking at runtime, DOORS has many applications beyond interoperability solutions, such as software development tools based on interpreters, debuggers and monitors that can dynamically interpose on objects.

An interrupt-driven approach can also be used for interoperability using the I/O Handling subsystem comprising the I/O Handler and its virtual devices.

The principle of the Wrapper Façade pattern [8] is applied in the design of the virtual devices, by encapsulating low-level functions for manipulating TCP/IP and UDP/IP sockets, UNIX pipes and files using a more concise, uniform and understandable OO class interface.

Basically each virtual device monitors one UNIX file descriptor, and the I/O Handler uses the select() UNIX system call on these descriptors. Upon detecting an event on any of these descriptors, the I/O Handler requests for an execution turn from the scheduler and passes control over to the responsible virtual device which then performs event-processing actions. This principle of operation of the I/O Handler used in DOORS closely corresponds to the Reactor pattern [9].

Thus, apart from using virtual devices for sockets, pipes and files, for this manner of communication,

user-written virtual devices can also be utilized for customized event processing for external applications, assuming the external system supports this Reactor-based approach and exports its file descriptors to DOORS.

For legacy systems which have a reasonably low level of interworking, this technique offers two advantages: Firstly, it can be used to overcome issues of interworking or short-term integration without suffering any loss in functionality nor needing any large overhead. Secondly, this technique allows for the interworking of such frameworks with many kinds of distributed object models without being tied to one particular type of distributed technology [10].

The Reactor-based approach is gaining in popularity owing its simplicity of approach, with implementations available in several major projects such as the X11[11] as well as the Apache Cocoon Project[12], and ORBs such as Orbix[13], TAO[14] and Orbacus[15].

From the three above-mentioned interoperability approaches, one can immediately infer that a wide plethora of hybrid solutions are implementable. This leads to many different types of possible usage. For example, because of the inheritance from the Protocol task class, CORBA-based servants in DOORS can potentially also serve as protocols in a protocol stack or event-based applications which use sockets.

One can easily envision scenarios such as an LDAP server task in DOORS which, using only pure CORBA calls, can serve LDAP requests to external CORBA clients interested in browsing directories, leading to implementations such as LDAP-enabled Implementation Repositories or a federation of CORBA Trading Servers.

4. EXAMPLE IMPLEMENTATION

One area in which DOORS has been employed is in the implementation of a unicast and multicast client and server for streaming experiment using MPEG 1 Layer III (mp3) audio.

The initial server prototype used the RTP and RTCP protocol modules for transporting the audio data over UDP sockets. The RTP frame is structured in such a way that the first part contains a 16-byte long RTP Header, which contains valuable information for stream synchronisation such as sequence numbers and timestamp values. This is then followed by the MPEG header and the MPEG frame which are created by an MPEG encoder. Every RTP frame contains exactly one MPEG frame as its payload.

Because RTP is a common way to multicast real-time multimedia, there are several clients already available for receiving and playing RTP streams, audio and video. For the purpose of testing this initial prototype, the audio server was multicasting packets with a TTL value of 16, from a Linux-based PC, and the streams were successfully received remotely with Linux- and windows-based RTP-aware mp3 players, such as FreeAmp[16].

This example was then expanded to include a client-server thread-based model which retains streaming via RTP whilst implementing a CORBA-based stream control mechanism for both point-to-point and point-to-multipoint connections as shown in Figure 8. The initial model uses the RTP implementations supplied by DOORS, but the CORBA functionality is provided via static interfaces with MICO [17]. This will be subsequently modified to use the dynamic interfaces provided with DOORS. This is a lightweight implementation of the model prescribed for the control and management of audio/video streams using CORBA Objects, by the OMG CORBA Telecommunications Domain Task Force [18].

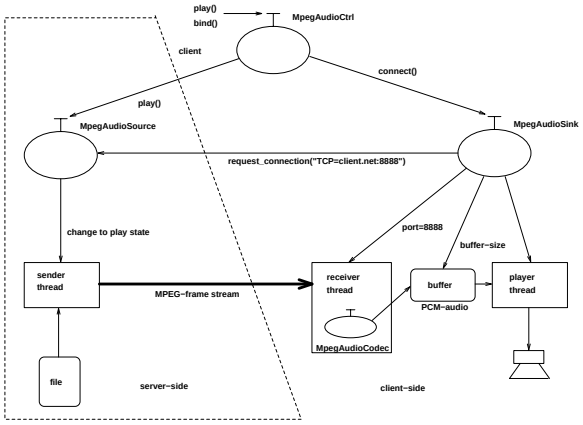

Figure 8. Audio Streaming Experiment

Three CORBA objects are used: *MpegAudioSource*, *MpegAudioCtrl* and *MpegAudioSink* representing the server object, control object and a client object, respectively. When a client wishes to open a connection to the server, it creates an MpegAudioCtrl object and an MpegAudioSink object and obtains their object references. It then obtains an object reference to the MpegAudioSource object, after which the client invokes a bind() operation on MpegAudioCtrl giving the references of the source and sink objects to bind them together.

The bind() operation calls connect() on the client object, which launches a receiver thread to start waiting for a connection from the server. At the same time, the client also launches a player thread.

When the receiver thread is ready, connect() calls request_connection() on the server-object with a parameter that gives the server an indication of the protocol and address to use. Figure 8 shows the client requesting the server to use TCP to connect to port 8888 on its host, “client.net”.

The server then launches a sender thread that first tries to open a connection to the client with the supplied parameters. If the connection is successfully opened, request_connection() returns positive result to connect() on the client object which, in turn, signals bind() on the control object about the successful connection.

Once a connection between client and server objects is established, the user can start controlling the MPEG-stream to be received. There are methods like play(), stop(), get_list() and select_item() for starting and stopping the stream, getting a list of streaming objects (mp3 filenames here) and selecting a streaming object, respectively.

An *MpegAudioCodec* object is used as an MPEG-audio decoder, which decodes MPEG frames and writes decoded PCM audio frames to a buffer that is read by the client's player thread. Because the receiver and player threads execute independently, with their own connections to the frame buffer, the buffer writing and reading methods remain thread-safe. *MpegAudioCodec* was created for easy encoding and decoding, and it uses functions provided by Lame[19], a freely available program for encoding and decoding MPEG audio.

5. DESIGN DECISIONS

DOORS is a framework that presents a model in which the complexity is hidden via base classes and modularly designed components, all of which interact together to provide uniform interfaces at key access points. It is a long-term project that will be under continual active development, with more component modules being added in, such as a basic CORBA Naming Service and an Interface Repository, to name a few. The code is available for public download and the code repository can be browsed online using any common web browser [20].

Design patterns, as documented in many instances, are highly effective in producing systems with a good design foundation rapidly. However, their implementations must be flexibly modified and adapted for event-based systems as well as to conform to a certain level of performance, scalability and distribution. Documentation of reusable methodologies in implementing common but important activities in protocol engineering are also needed, such as connection management, peer abstraction and service access points.

The benefits seen when collections of patterns work together are enormous. In addition to those previously mentioned, others include Singleton Factories for protocols, Flyweight[3] Singletons acting as CORBA servants or Singletons and State Machines flexibly employing various types of garbage collection strategies. However, proper care must also be exercised in certain combinations. For example, if a few Singletons are attempting to construct one other, without proper error and exception handling, the result may be dangerous deadlocks, race conditions or improperly constructed objects.

Also when it comes to interoperability issues, at times it becomes necessary to obtain more information empirically owing to a lack of well-specified standards. One such example is the usage of the CORBA Interoperable Object References (IOR) used in IIOP. Since there is no byteorder information in IOR according to the specifications, it seems that many ORB implementations actually insert an extra four byte long field to the stringified form of an IOR in order to deal with the situation of the IOR publisher having a different byte order than the one using a stringified IOR. Also, in some ORBs, the major and minor version information fields in the IOR seem to be encoded as unsigned short types, rather than being octets, as specified.

Much of the techniques presented could have used alternatives such as synchronous multithreading mechanisms. However, threads could lead to high performance overheads and require a deep knowledge of synchronization patterns and principles in order to manage access to shared resources. Threading may also not be available on all platforms. Hence the design methodologies in the component libraries of DOORS harness the usage of the design patterns and their modifications thereof, to achieve the maximum portability possible for independence from underlying operating systems. At the same time, DOORS does not limit the developer to designing only single-threaded applications atop its component libraries, should a need for threads does arise, as shown by the streaming experiment in section 4.

6. CONCLUSIONS

The Internet as well as the telecommunications domain are rich with communication methods which generally appear as discrete events occurring at many levels of granularity, ranging from deep within the detailed implementations of network components to the way those network components intelligently interact with each other.

The convergence of communications, computing and contents is catalyzing the development of a new breed of applications, and DOORS remains a highly suitable platform for designing, prototyping and implementing various kinds of discrete event-based systems that can provide a basic functionality for such applications.

REFERENCES

[1] OMG: *"The Common Object Request Broker: Architecture and Specification. CORBA*

V2.3.1”. October 1999.

[2] J. Harju, B. Silverajan, I. Toivanen: *"OVOPS – Experiences in Telecommunications Protocols with an OO Based Implementation Framework”*. Proc. ECOOP ´97 Workshop on Object Oriented Technology for Telecommunications Services Engineering, Jyvaskyla , Finland June 9 - 13, 1997.

[3] E. Gamma, R. Helm, R. Johnson, J. Vlissides: *"Design Patterns, Elements of Reusable Object-Oriented Software”*. Addison-Wesley 1995.

[4] IETF RFC1889 *"RTP: A Transport Protocol for Real-Time Applications”*.

[5] TINA Consortium, http://www.tinac.com

[6] OMG : *"Messaging Service Document orbos/98-03-12”*.

[7] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal: *"Pattern-Oriented Software Architecture: A System of Patterns”*. John Wiley & Sons, Inc., New York, 1996.

[8] Douglas C. Schmidt: *"Wrapper Facade: A Structural Pattern for Encapsulating Functions within Classes”*. C++ Report, SIGS, Vol. 11, No 2, February, 1999.

[9] Douglas C. Schmidt: *"Reactor: An Object Behavioural Pattern for Concurrent Event Demultiplexing and Event Handler Dispatching”*. Pattern Languages of Program Design (J.O. Coplien and D. C. Schmidt, eds.), Reading, MA: Addison-Wesley, 1995.

[10] B. Silverajan, J.Harju: *"Enhancing an Event-Based OO Framework for Distributed Programming”*. Proceedings of TOOLS USA '99, Santa Barbara CA August 1 - 5, 1999. pp 162 - 171, IEEE Computer Society ISBN 0-7695-0278-4

[11] X.org : X11R6 Specifications, http://www.x.org

[12] The Apache Cocoon Project, http://xml.apache.org/cocoon/

[13] IONA Technologies : Orbix 2.2 Reference Guide, March 1997.

[14] TAO (The Ace ORB), http://www.cs.wustl.edu/~schmidt/TAO.html

[15] Object Oriented Concepts, Inc. : ORBacus, http://www.ooc.com

[16] FreeAmp : Free Audio Music Player, http://www.freeamp.org

[17] MICO (MICO Is CORBA), http://www.mico.org

[18] OMG CORBAtelecoms : *"Telecommunications Domain Specifications, Version 1.0”*. June 1998

[19] The LAME Project, http://www.sulaco.org/mp3/

[20] DOORS, https://garuda.atm.tut.fi/doors/