Introducing SEAN: Signaling Entity for ATM Networks

Sean Mountcastle * David Talmage * Bilal Khan * Spencer Marsh †
Abdella Battou † Daniel C. Lee ‡

Abstract

SEAN is freely distributed, object–oriented, extensible software for research and development in host ATM signaling. SEAN includes a complete source-level release of the host native ATM protocol stack, and implements the ATM User Network Interface, compliant to the ITU Q.2931 specification for point-to-point calls, the ITU Q.2971 extension for point-to-multipoint calls, and the ATM Forum extension UNI-4.0 for leaf initiated join calls. SEAN provides APIs to the programmers writing application programs that require ATM signaling. Developers can easily modify and extend SEAN, using the framework library released together. This paper briefly describes essential parts of SEAN's architecture and guides the users and protocol developers.

1 Introduction

In recent years we have witnessed a remarkable growth in the availability of affordable ATM hardware. Yet the majority of the scientific research community has found it difficult to engage in much needed basic research in the areas of ATM protocol design and network performance optimization. We believe that this has been partly due to the dearth of affordable, publicly available software implementations of the host ATM protocol stack. In response to this situation, the U.S. Naval Research Laboratory has released SEAN (Signaling Entity for ATM Networks)[1]. SEAN is a freely distributed, extensible environment for research and development in host ATM signaling. SEAN includes a complete source-level release of the host native ATM protocol stack, and implements the ATM User Network Interface, compliant to the ITU Q.2931 specification for point-to-point calls, the ITU Q.2971 extension for point-to-multipoint calls, and the ATM Forum extension UNI-4.0 for leaf initiated join calls. SEAN compiles and runs on SunOS 5.x (Solaris). It is known to compile on SunOS 4.x, Linux 2.x, and NetBSD 1.3.x. It is our hope that SEAN will serve as the starting point for bold new ATM initiatives, both in terms of probing academic research and innovative commercial development.

SEAN software is object–oriented and can be used both for experimentation/operation and simulation. SEAN's host native ATM protocol stack can be installed in the host to perform experiments. SEAN also includes software that simulates an ATM switch. SEAN supports point to point calls, signaling of individual QoS parameters, ABR signaling for point to point calls, ATM multicast signaling, ATM anycast, traffic parameter negotiation, etc. The SEAN architecture includes a few essential parts - the signaling daemons (SD and SIM models), the application programming interface (API), and the address registration daemon (ILMID).

The **SD** is the live signaling daemon, whose implementation includes a driver interface layer, SSCOP, SSCF, the signaling layer (Q.2931, Q.2971, and UNI-4.0), the Call Control layer, etc. Figure 1 shows several applications communicating with one another on an ATM network through their APIs and Signaling Daemons.

The **SIM** is the simulated signaling daemon. The motivation behind the inclusion of a simulated signaling daemon is twofold; to enable research in ATM signaling even in the absence of ATM hardware, and to allow researchers to debug their native ATM applications "offline", i.e. off the network, to ensure proper behavior prior to running them live. The simulated signaling daemon models an arbitrary number, n, of signaling-capable hosts, each connected by a lossless link to a different port of a common n-port switch. Thus the simulated signaling daemon internally instantiates n user-side UNI stacks, and n network-side UNI stacks, each of which is quite similar to the single stack made in the SD. Figure 2 illustrates that a user can use SEAN's simulation module in the absence of ATM hardware.

The **API** is a set of C++ classes that provides access to ATM network services to applications that require ATM signaling. The SEAN API separates an application program from the ATM signaling services that it requires. It hides the signaling mechanisms from the application. The

* ITT Industries, Advanced Engineering & Sciences, Advanced Technology Group, at the Center for Computational Sciences of the Naval Research Laboratory, Washington D.C.
† Princeton Networks Inc.
‡ University of Southern California, Department of Electrical Engineering - Systems, 3740 McClintock Avenue, Los Angeles, CA 90089-2565. dclee@usc.edu

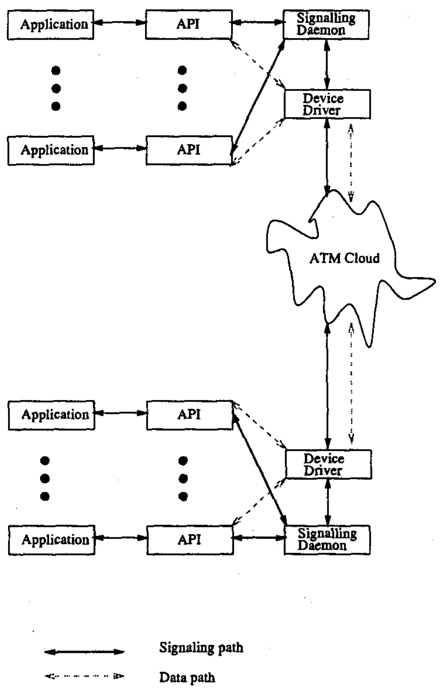

Figure 1: Application layer entities communicating through SEAN and ATM network

SEAN API implements an object-oriented interface to non-blocking functions for creating native ATM applications.

The **ILMID** (Integrated Local Management Interface Daemon), provides an interface to each ATM device for status information, parameter configuration, and address registration. It uses the SNMP[2] protocol for communication with its peer. The **ILMI** protocol implemented in **ILMID** provides a 2-way exchange of parameters, configuration information, and address registration across an ATM User Network Interface (**UNI**) between a host and a switch. As such, it is a subset of the full protocol specified by the ATM Forum specification[3].

The SEAN binaries of those components (SD, SIM, ILMI, API library) may be used immediately, as is, to conduct research experiments in ATM signaling. The binaries of SD and ILMI require a suitable ATM network interface card to operate. However, the binary of SIM can be used without ATM hardware. API library routines for developing native ATM applications work (transparently) on any host that is running either SD or SIM binary.

SEAN also includes many lower-level components that are of interest to researchers who wish to extend the set of standard UNI features supported by SEAN, or who are

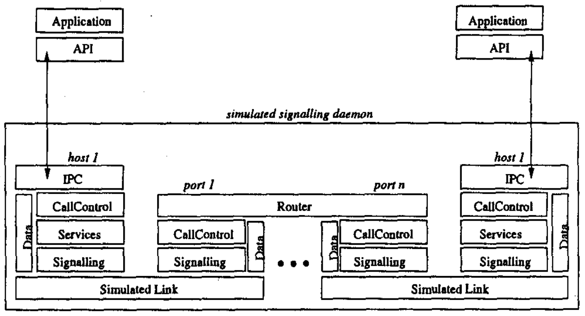

Figure 2: Application layer entities communicating through SEAN's simulator

conducting experiments that require nonstandard modifications to the ATM signaling protocols. These include

- the call-control finite state machine, for user and network side protocol stacks
- the user and network side of the UNI-4.0 finite state machine
- encoders and decoders of UNI-4.0, UNI-3.1 packets
- encoders and decoders of UNI-4.0, UNI-3.1 information elements
- the UNI-SSCF coordination finite state machine
- the SSCF finite state machine
- the SSCOP finite state machine
- AAL wrapper encapsulating all data interaction with drivers
- a framework library for rapid prototyping of communication protocols and simulations

2 SEAN architecture

2.1 The Live Signaling Daemon (SD)

The **SD** is the live signaling daemon. It is implemented as a stack of six layers (see figure 3) implemented over a communication framework library. Bottom up, the stack includes a driver interface layer, the QSAAL layer (SSCOP and SSCF), the signaling layer (Q.2931, Q.2971, and UNI-4.0), the Service Registration layer, the Call Control layer, and finally the API interface layer.

1. The AAL part of the QSAAL is commonly implemented in hardware and thus lives in the driver. The Driver interface binds the (0,5) signaling connection to a file descriptor used to read and write SSCOP protocol data units.
2. The SSCOP layer provides a reliable transport for the signaling messages between two peer signaling layers.

3. The signaling layer implements the ITU Q.2931 specification for point-to-point calls, the ITU Q.2971 extension for point-to-multipoint calls, as well as the ATM Forum extension UNI-4.0 for leaf initiated join calls.
4. The service registration layer allows applications to specify call profiles to be matched against incoming calls (intended for server applications), as well as as call profiles for leaf-initiated-joins.
5. The call control provides routing, call admission control logic, resource management (such as VPIs, VCIs, and call reference values).
6. Finally, there is the SD–side IPC (inter–process communication) for the signaling session between SD and applications written by using the SEAN API.

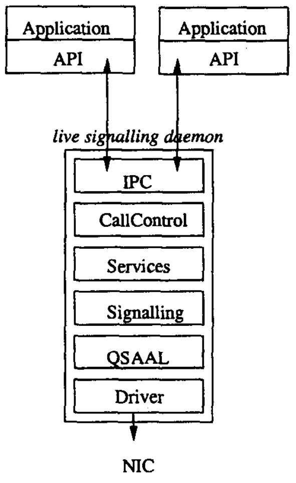

Figure 3: The live signaling daemon (SD)

2.2 Simulated Signaling Daemon (SIM)

The simulated signaling daemon models an arbitrary set of n signaling-capable hosts, each connected by a lossless link to a different port of a common n-port switch. Thus the simulated signaling daemon internally instantiates n user-side UNI stacks and n network-side UNI stacks, each of which is quite similar to the single stack made in the SD. Because the simulated signaling daemon assumes lossless links between the switch and its n hosts, each of the $2n$ UNI stacks lack a QSAAL layer and does not ever communicate with an ATM driver. Furthermore, the n network-side UNI stacks do not instantiate a service registration layer and have a common routing module in place of the IPC layer.

2.3 Application Programming Interface

The SEAN API implements an object-oriented interface to non-blocking functions for creating native ATM applications. The API is based on the call-back model. Each call-back function is atomic. That is, it is not interrupted by any event even if one of its own actions caused that event. The SEAN API separates an application program from the ATM signaling services that it requires. It hides the signaling mechanisms from the application. It alerts the application to a change in a call, service, or leaf through a collection of call-back functions. Client and server applications communicate with a Signaling Daemon (SD). They tell the SD when they want to establish or tear down a call, register or de-register a service, or join or drop a leaf. The SD performs all of the ATM signaling and informs them when a call, leaf, or service becomes active or inactive. In addition, the SD informs them when there is data for a call to read. All of this happens asynchronously and atomically through call-backs. Figures 4 and 5 illustrate the interactions between API objects and the SD or SIM. There is also a way for applications to schedule periodic self-events inside applications, and this does not require the services of an SD. There are five kinds of SEAN API objects: ATM_Interface, Controller, ATM_Call, ATM_Leaf, and ATM_Service.

An ATM_Interface object represents a logical network interface. An ATM_Interface connects a SEAN application to logical ATM interface of a physical ATM network interface card. The ATM_Interface encapsulates a "signaling session" with an SD and a "data_session" with the driver. In an application using SEAN, there must be at least one, each representing a different logical network interface and therefore a different ATM address. There is at most one ATM_Interface object per logical network interface. Function Open_ATM_Interface(const char* devname, int ccdport) is provided for programmers. This function instantiates an ATM_Interface object and makes it reference the special file "devname" and an inter-process–communication port. The API object communicates with SD through this port. The special file is to communicate the data messages with the driver of the NIC (network interface card).

Each ATM_Call, ATM_Leaf, and ATM_Service object implements a call, leaf of a point-to-multi-point call, or service, respectively. Each is described by a collection of attributes that correspond to the UNI 4.0 information elements. Examples of the information elements are calling party number, called party number, quality of service parameter, traffic descriptor, broadband bearer capability, AAL parameter, etc. ATM_Attributes class is used to express the attributes. ATM_Attributes class has, along with others, its private variables,

```
InfoElem* _ie[num_ie];
```

describing UNI information elements. The ATM_Attributes has methods (member functions) for setting and getting these variables. ATM_Service, ATM_Call, ATM_Leaf are in fact subclasses of ATM_Attributes class, so each inherits the variables representing the information elements from ATM_Attributes class. The ATM_Attributes of an ATM_Service describe the information elements that must be present in call Setup messages to the service. The Setup may have more information elements than the ATM_Service specifies. The ATM_Attributes of an Outgoing_ATM_Call (a subclass of ATM_Call) are the information elements that are present in the Setup message that initiates the call. The ATM_Attributes of an Incoming_ATM_Call (a subclass of ATM_Call) are the information elements that were present in the Setup message that initiated the call. ATM_Call and ATM_Service are like a key and the lock that it fits. The ATM_Attributes of an ATM_Service enumerate the ATM_Attributes that the service expects its callers to have. The Signaling Daemon connects incoming calls to services based on how well the sets of attributes match. Each of these classes (ATM_Call, ATM_Service, ATM_Leaf) has an associated API_FSM object that represents a finite state machine encapsulating the state of the call, service, or the leaf. For example, ATM_Call object is in one of the following states;

```
CCD_DEAD,
USER_PREPARING_CALL,
CALL_WAITING_FOR_LIJ,
INCOMING_CALL_WAITING,
OUTGOING_CALL_INITIATED,
CALL_P2P_READY,
CALL_P2MP_READY,
CALL_LEAF_JOIN_PRESENT,
CALL_RELEASE_INITIATED,
CALL_RELEASED,
LIJ_PREPARING,
LIJ_INITIATED.
```

ATM_Service object is in one of the following states;

```
CCD_DEAD,
USER_PREPARING_SERVICE,
SERVICE_REGISTER_INITIATED,
SERVICE_DEREGISTER_INITIATED,
SERVICE_ACTIVE,
INCOMING_CALL_PRESENT,
SERVICE_INACTIVE.
```

ATM_Leaf object is in one of the following states;

```
CCD_DEAD,
USER_PREPARING_LEAF,
INCOMING_LIJ,
ADDPARTY_REQUESTED,
DROPPARTY_REQUESTED,
LEAF_READY,
LEAF_RELEASED.
```

Each of these objects is associated with one and only one ATM_Interface. Each of these objects is instantiated and goes through certain state transitions in response to invocations of its member functions or in response to message arrivals from SD. Also, each of these objects is associated with one and only one Controller. (Also, each ATM_Interface is associated with one or more Controller objects.)

A Controller object is a call and service manager. Every SEAN application has at least one. Controllers place ATM calls and register services so that they can receive ATM calls. Controllers also react to events that happen at the ATM_Interfaces. These events may be ones that pertain to establishing and tearing down calls (such as the accepting an incoming call or noticing the acceptance of a call by the other party), registration of a service, or a lower-level events such as data arrival in an already established call. There are twelve such events formally specified. The base class Controller has 12 pure virtual methods corresponding to the 12 API events as shown in Table 1. The twelve events that happen at the ATM_Interfaces

Event	Callback
Boot	`Boot()`
Call Active	`Call_Active()`
Call Inactive	`Call_Inctive()`
Incoming Call From Root	`Incoming_Call_From_Root()`
Incoming Leaf Initiated Join	`Incoming_LIJ()`
Leaf Active	`Leaf_Active()`
Leaf Inactive	`Leaf_Inctive()`
Service Active	`Service_Active()`
Service Inactive	`Service_Inctive()`
Incoming Call	`Incoming_Call()`
Incoming Data	`Incoming_Data()`
Periodic Callback	`PeriodicCallback()`

Table 1: Events and their call-backs

are in fact events that happen in ATM_Call, ATM_Leaf, ATM_Service objects. (The ATM_Interface object encapsulates the finite state of all the calls and the services of the application that goes through the network interface. A single ATM_Interface object may hide the finite state machines of many ATM_Call and ATM_Service instances.) The API is designed in such a way that when a call, leaf, or service event occurs, the corresponding Controller Callback method is called. Thus, the Controller Callback functions written by the application programmer determine how those events drive the program. (The application programmer may evoke any one of the public methods of the ATM_Call, ATM_Service, or ATM_Leaf.)

The API uses three different modules to achieve its functionality. First, it uses a lower-level codec library to encode and decode ITU and ATM Forum specified information elements and messages. Second, it uses the IPC library to communicate over a Unix Domain socket connection for a signaling session with the SD. Third, it uses the driver interface to bind the (vpi,vci) pair identifying an ATM connection to a file descriptor so that data can be sent and received on that connection. These three modules are isolated well, using a base interface that delegates to actual implementations.

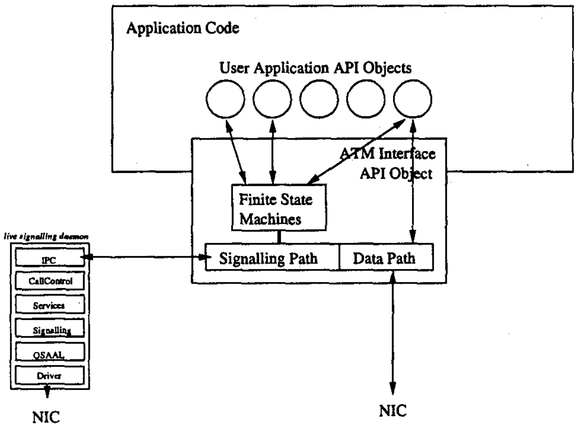

Figure 4: API interaction with the live signaling daemon (SD)

3 User Guide

3.1 Writing application programs

It is easy to write a SEAN program. Follow these simple steps:

1. Instantiate the ATM_Interface objects.
2. Write the Controller objects. For each controller object, the programmer must write the controller's callback virtual functions. Then, instantiate the controller.
3. Create associations between the ATM_Interfaces and the Controllers so that the ATM_Interfaces can notify the Controllers of the API events that pertain to them. `ATM_Interface` class' member function `Associate_Controller` creates the association for the programmer.
4. Call SEAN_Run() function.

We provide the following example.

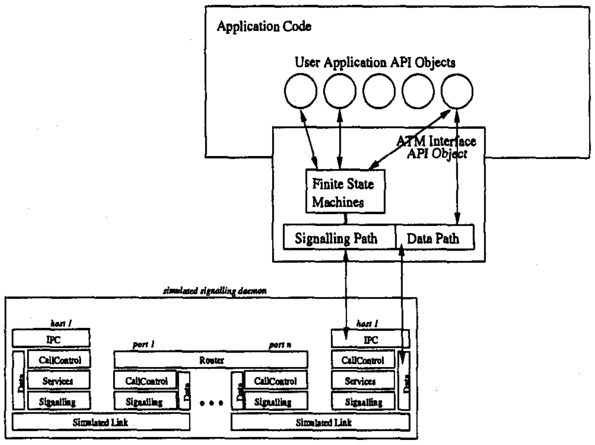

Figure 5: API interaction with the simulated signaling daemon (SIM)

```
#include <sean/api/ATM_Interface.h>
int main(int argc, char **argc)
{
  ...
 int port = atoi(argv[1]);
// Create ATM_Interfaces
 ATM_Interface &interface =
  Open_ATM_Interface("/dev/una0", port);
  ...
// Create Controllers. (BaseService is
// a subclass of Controller.)
 BaseService *service = 0;
 service = new BaseService(interface);
  ...
// Associate the Controllers with
//  their ATM_Interfaces
 interface.Associate_Controller(*service);
  ...

 SEAN_Run();
 exit(0);
}
```

Once the ATM_Interfaces and the Controllers have been created and the associations between them have been made, a SEAN application must start the API by calling `SEAN_Run()`.

3.2 Running SD or SIM, application programs

The Signaling Daemon, SD, performs all UNI signaling tasks for the SEAN applications that talk to it. It listens on an arbitrary but "well known" port for requests from SEAN API applications. SD requires one command line argument, the number of the well known port. For example, to run SD on a Solaris computer and make it listen on port 9000, a user commands (with executable name `sd_SunOS5`):

```
sd_SunOS5 9000
```

SIM models a switch connected to arbitrary number of hosts. Specifically, each of the simulated switches ports is connected to a host stack similar to the one implemented inside the live signaling daemon. Each of these host stacks listens on a socket for incoming connection. Command for running SIM requires two command line arguments; for example, with executable name simswitch the command line may be

```
simswitch base-ipc-port host-file
```

where

base-ipc-port is the number of the first socket. In a simulated switch of n ports, the first port signaling stack listens on the port $base - ipc - port$. The second listens on $base - ipc - port + 1$. The n^{th} port listens on $base - ipc - port + n - 1$.

host-file is the name of the file that contains the names and ATM addresses of the hosts in the simulation. Each line of the file names one host and its ATM address. The ATM address comes first on each line. The host name comes second. Comments begin with a # and end at the end of the line. Blank lines are ignored. A sample host file, with file name host.file, is included below:

```
#                      Nsap                        host
0x47.0005.80.ffde00.0000.0000.0104.000000000000.00 foo
0x47000580ffde00000000000104000000000001.00        bar
47.0005.80.ffde00.0000.0000.0104.000000000002.00   bas
```

After the execution of:

```
simswitch 9000 host.file
```

applications may initiate a signaling sessions to ports 9000-9002, and operate indistinguishably from if they were running of a live signaling daemon (sd). Note that all applications must run on the same host as the simswitch.

When SD's are running in different hosts, applications can start running in the hosts. Or, it SIM is running in a host, multiple applications can run in the host and communicate with each other.

4 Extending and Modifying SEAN: Developer Guide

As SEAN's source code is released, SEAN can be modified and extended for further research and development. In fact, the object–oriented feature makes it convenient. A C++ framework[4, 5, 6, 7] library, CASiNO (Component Architecture for Simulating Network Objects)[8] has been used for modular implementation of SEAN. CASiNO provides programmers with a rich yet modular, coarse-grained data flow architecture, with an interface to a reactor Kernel that manages the application's requirements for asynchronous notifications of I/O, real timers, and custom intra-application interrupts. These features enable developers to write applications that are driven by both data flow and by asynchronous events, while allowing them to keep these two functionalities distinct. CASiNO has been released for the public with SEAN. A developer can use CASiNO to extend SEAN and/or to implement other network protocols.

References

[1] Naval Research Laboratory, "SEAN: Signalling entity for ATM networks." http://www.nrl.navy.mil/ccs/project/public/sean/SEAN-dev.html.

[2] M. T. Rose, *The Simple Book: an Introduction to Internet Management*. Englewood Cliffs, New Jersey: PTR Prentice Hall, second ed., 1994.

[3] ATM Forum Technical Committee, "Integrated local management interface (ILMI) specification." Version 4.0n af-ilmi-0065.000, September 1996.

[4] H. Hüni, R. Johnson, and R. Engel, "A framework for network protocol software," in *Annual ACM Conference on Object–Oriented Programming Systems*, 1995.

[5] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Pattern, Elements of Object–Oriented Software*. Reading, MA: Addison–Wesley, 1995.

[6] L. P. Deutsch, "Design reuse and frameworks in the smalltalk–80 system," in *Software Reusability, Volume II: Applications and Experience* (T. J. Biggerstaff and A. J. Perlis, eds.), (Reading, MA), Addison–Wesley, 1989.

[7] R. E. Johnson and B. Foote, "Designing reusable classes," *Journal of Object–Oriented Programming*, vol. 1, pp. 22–35, June/July 1988.

[8] S. Mountcastle, D. Talmage, S. Marsh, B. Khan, A. Battou, and D. C. Lee, "CASiNO: A component architecture for simulating network objects," in *Proceedings of 1999 Symposium on Performance Evaluation of Computer and Telecommunication Systems*, (Chicago, IL), pp. 261–272, Society for Computer Simulation International, July 1999.