SOFTWARE—PRACTICE AND EXPERIENCE
Softw. Pract. Exper. 2002; **32**:1099–1128 (DOI: 10.1002/spe.475)

CASiNO: Component Architecture for Simulating Network Objects‡

Abdella Battou[1], Bilal Khan[2], Daniel C. Lee[3,*,†], Spencer Marsh[1], Sean Mountcastle[2] and David Talmage[2]

[1]*FirstWave Intelligent Optical Networks, Inc., Greenbelt, Maryland 20770, U.S.A*
[2]*ITT Industries, Advanced Engineering & Sciences, Advanced Technology Group, Center for Computational Sciences of the Naval Research Laboratory, Naval Research Laboratory, Code 5591, Washington D.C., 20375, U.S.A.*
[3]*University of Southern California, Department of Electrical Engineering, 3740 McClintock Avenue, Los Angeles, CA 90089-2565, U.S.A.*

SUMMARY

We describe the Component Architecture for Simulating Network Objects (CASiNO) useful for the implementation of communication protocol stacks and network simulators. This framework implements a rich, modular coarse-grained dataflow architecture, with an interface to a reactor kernel that manages the application's handlers for asynchronous I/O, real timers and custom interrupts. These features enable developers to write applications that are driven by both data flow and asynchronous event delivery, while allowing them to keep these two functionalities distinct. We provide an example program and expository comments on the program to illustrate the use of the CASiNO framework. Published in 2002 by John Wiley & Sons, Ltd.

KEY WORDS: telecommunications; protocols; object orientation; framework; simulation

1. INTRODUCTION

A framework is generally defined as a set of cooperating classes that make up a reusable design for a specific class of software [1–3]. 'Frameworks are becoming increasingly common and important. They are the way that object-oriented systems achieve the most reuse' [1]. In this paper, we present a design and C++ implementation of the CASiNO (Component Architecture for Simulating

*Correspondence to: Daniel C. Lee, University of Southern California, Department of Electrical Engineering, 3740 McClintock Avenue, Los Angeles, CA 90089-2565, U.S.A.
†E-mail: dclee@usc.edu

Published in 2002 by John Wiley & Sons, Ltd.

Received 27 September 2001
Revised 26 April 2002
Accepted 26 April 2002

Network Objects) framework, which has been used for modular implementation of several complex communication protocols and network simulators. Some of the projects built using CASiNO include:

(1) SEAN [4] (Signaling Entity for ATM Networks), which implements signaling as part of the user-to-network interface for Asynchronous Transfer Mode (ATM) networks;
(2) PRouST [5] (PNNI Routing Simulation Toolkit), which provides hierarchical definition and routing as part of the network-to-network protocols of ATM networks;
(3) TRON [6] (Toolkit for Routing in Optical Networks), which provides an implementation of a modified Open Shortest Path First (OSPF) protocol for Wavelength Division Multiplexing (WDM) all-optical networks.

These projects were developed at the Naval Research Laboratory. Although each system was quite complex, the underlying CASiNO framework provided the following.

(1) Ease of design. Implementations of protocols that are carried out 'from the ground up' (e.g. in a programming language like C) require substantial effort to implement common communication constructs, such as timers, message queues, etc. Given a general-purpose language like C, a programmer can build these in many different ways. A framework, on the other hand, assumes responsibility for all the common low-level aspects of the protocol's implementation, while providing high-level abstractions with which a programmer can quickly specify the logical structure of the protocol. A framework reduces the number of ways in which the programmer can implement a protocol without sacrificing the range of what the programmer can accomplish. Implementations that use a framework are more 'uniform' in their design, because they have a common structure imposed on them. This structure makes the design and maintenance process simpler.
(2) Ease of description. By providing common high-level abstractions with which protocols can be quickly implemented, a framework also implicitly provides a succinct language by which to describe and document the design of protocol software.
(3) Ease of maintenance. Because protocol implementations over a framework are easy to describe in terms of high-level abstractions, it is easier to maintain modular communication protocol software.
(4) Portability. A framework hides all system-dependent operations, exposing only the high-level abstractions needed for protocol development. Porting a protocol to a new platform amounts to porting the framework itself to the new platform. This process only needs to be carried out once; subsequently, any protocols implemented using the framework will also operate on the new architecture.
(5) Longevity of code. Because implementations over a framework are easy to describe using the high-level abstractions, the learning curve for implementations is also relatively short. Projects are therefore less susceptible to personnel and hardware changes. SEAN, PRouST, TRON and other CASiNO-based software projects have served as a starting point for the work of other researchers. For example, at the Applied Research Laboratory at Washington University, Turner and Wu successfully combined PRouST and SEAN with their Gigabit switch controller to form a network node [7].
(6) Visualization. Because the code is written using high-level abstractions, the invocation of these abstractions can be logged at run-time. Such logs can then be visualized or analyzed statistically.

CASiNO consists of two distinct halves: the Data-Flow Architecture and the Reactor Kernel. The *Data-Flow Architecture* is the name given to the collection of classes that are used to define the behavior of different modules within the protocol stack that is being implemented. The Data-Flow Architecture is designed to maximize code reuse, make isolation of components easy and to render the interactions between modules as transparent as possible. The *Reactor Kernel* is the name given to the collection of classes that provide applications with access to asynchronous events, such as the setting and expiration of timers, notification of data arrival and the delivery and receipt of intra-application interrupts. In Sections 3 and 4, we introduce components of CASiNO. In illustrating their use, we demonstrate the needs and characteristics of applications developed using CASiNO. Although the focus is on the abstract base classes that form the foundation of CASiNO, from time to time we comment on the necessary requirements placed on programmers using this library. Also, for a better perspective, we sometimes mention abridged details of CASiNO's internal mechanisms and code.

2. RELATED WORKS

CASiNO is based on ideas shown in the Conduits+ [8,9] framework. A notably different feature of CASiNO is its parallel hierarchies of Behavior and Actor classes, which will be described in Section 3. The Jacob [10] framework has a commonality with CASiNO, in that both are based on Conduits+. In addition to these historically related frameworks, it is worth comparing CASiNO with a currently popular network simulator ns, [11,12]. If CASiNO is used to build a network simulator, the resulting simulator and the widely used network simulator, ns, have some commonalities in the simulator architecture. ns' counterpart of CASiNO's Reactor Kernel would be ns Scheduler. ns Scheduler runs by selecting and executing the next earliest event scheduled, as does CASiNO' Reactor Kernel. One of the differences between the ns and CASiNO codes is that ns, while written in C++, has an OTcl [13] interpreter as a front-end. ns supports a class hierarchy in C++ and a similar class hierarchy within the OTcl interpreter, thus allowing split-level programming [11]. Another difference may be that the CASiNO framework is actually designed for implementing a network protocol with an object-oriented software, in addition to being useful for developing a simulator. CASiNO itself is not a simulator, it is an object-oriented framework for developing a network protocol and a network simulator. Noting this, we believe that the Data-Flow Architecture of CASiNO may be easily be integrated to ns with minor modifications. (The software economy of such integration is an interesting topic to be investigated.)

3. THE DATA-FLOW ARCHITECTURE

An object-oriented application is a set of objects working together to achieve a common goal. These objects communicate by sending messages to one another (invoking methods of one another). In networking software [14–23], one can identify two categories of objects—those that form the architecture of the different protocols and those that form the packets being sent from entity to entity. We refer to the architecture objects as *Conduits* and the different packets as *Visitors* [8]. CASiNO provides Conduit and Visitor classes; software design using the Data-Flow Architecture centers around

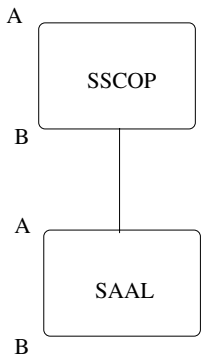

Figure 1. Connected Conduits.

the flow of Visitor objects through Conduit objects. (The term *Data-Flow* is used here to mean coarse-grained 'packet' flow, rather than classical fine-grained streams.)

A Conduit object typically implements a logical component within a network protocol. Conduits are connected together in an undirected graph to form what is called the protocol stack. While isolating its private information, a Conduit object has definite points of contact with other Conduits. Just as layers in a network protocol stack typically have interfaces with their immediate higher and lower layers, a Conduit has two sides, side A and side B, which are pointers to neighboring Conduits. In much the same way that data passes through each layer of the network protocol stack, so too within the a CASiNO application—a Visitor arriving from side A is acted on at a Conduit and then passed on out of side B. (Conversely, a Visitor arriving from side B might be passed on to side A after being acted on in this Conduit). In this sense, Conduits are bidirectional channels that act on transient data units, called Visitors. Just as two adjacent layers of the network protocol stack interface with each other, so too can Conduit objects be connected together. The programmer can connect instantiated Conduits by using Conduit class' friend functions:

```
Conduit * A_half(Conduit * c)
Conduit * B_half(Conduit * c)
Bool Join(Conduit * c1, Conduit * c2,
          Visitor * key1 = 0, Visitor * key2 = 0)
```

The architecture of two connected Conduit objects is illustrated by drawing a line between them. For example, Figure 1 illustrates that side B of the Conduit object named SSCOP is connected to side A of the Conduit object named SAAL. In this example, the SAAL [24] conduit provides convergence functions of the ATM Adaptation Later (AAL) in the control plane of the ATM model, while the SSCOP conduit implements the Service Specific Connection Oriented Protocol [25] which provides reliable transport for ATM control messages.

Example 1 illustrates how the programmer builds such a connected group of Conduits.

Example 1. Before connecting the two Conduits, the Conduits must be created. The following abbreviated C++ code instantiates and names them:

```
_sscop = new Conduit("SSCOP", sscop_protocol);
_saal = new Conduit("SAAL", aal_protocol);
```

Then, to connect side B of SSCOP to side A of SAAL, the friend functions are used as follows:

```
Join(B_half(_sscop), A_half(_saal));
```

The consequence of the above code is that any Visitor leaving out of side A of the Conduit SAAL will be passed into the side B of the SSCOP Conduit. Likewise, a Visitor leaving the SSCOP Conduit out of its side B will enter into the SAAL Conduit from its side A. Connecting Conduits in this manner determines possible *communication patterns between the objects during the application's lifetime. It should be noted, however, that the decision of whether to eject a Visitor out of a Conduits side A (or eject it out of the Conduits side B) to absorb the Visitor (or delay it), rests in the logic of the actual Conduit and Visitor. The connectivity of the Conduits does not make this decision, rather the geometric configuration of Conduit interconnections only limits the range of possible outcomes.*

Just as data units are processed in each layer of the network protocol stack (e.g. segmented, assembled, attached to control information, etc.), a Visitors passage through a network of Conduits may trigger similar actions. The general manner in which Visitors flow through a Conduit is referred to as the Conduits *behavior*. Instead of *sub-classing* the Conduit to specialize in the different behaviors, each Conduit *has a* Behavior. There are five derived types of Behavior: Adapter, Mux, Factory, Protocol and Cluster, based loosely on the concepts presented in [8]. Behaviors refer to generic semantic notions: an *Adapter* is a starting or ending point of a protocol stack; a *Mux* is a one-to-many/many-to-one branch point in a protocol stack; a *Protocol* is a finite state machine; a *Factory* is an object capable of instantiating new Conduits and joining them up; and a *Cluster* is a black-box abstraction of a sub-network of Conduits. Because Behaviors represent only general semantic notions, they must be made concrete by composing them with appropriate user-defined Actors; each Behavior *has an* Actor. For example, a Protocol behavior requires a user-defined state machine to be concretely defined. Instances of Conduits, Behaviors and Actors are always in one-to-one-to-one correspondence.

To summarize, CASiNO has three tiers in its design. (i) The Conduit level of abstraction represents building blocks of the protocol stack; the function of the Conduit class is to provide a uniform interface for joining these blocks together and injecting Visitors into the resulting networks. (ii) The Behavior level of abstraction represents broad classes, each capturing different semantics regarding the Conduits purpose and activity. (iii) At the Actor level of abstraction, the user defines concrete implementations. Ultimately, CASiNO application programmers must design a suitable Actor for each functionally distinct Conduit they intend to instantiate.

As an implementation detail, all Conduits, Behaviors and Actors must be allocated on the heap. (The implication of this to the programmer is that the 'new' operator is used for creating an actual object of these classes. However, the pointer to these objects can be local or global variables, so statements like 'Conduit * pointer_name' may be made.) Deletion of a Conduit, automatically deletes any associated Behavior and Actor objects—the destructor of the Conduit eventually deletes them. Because the objects have identical lifetimes, the relation between these objects is more of 'aggregation' than 'acquaintance' or 'association' [1, p. 23].

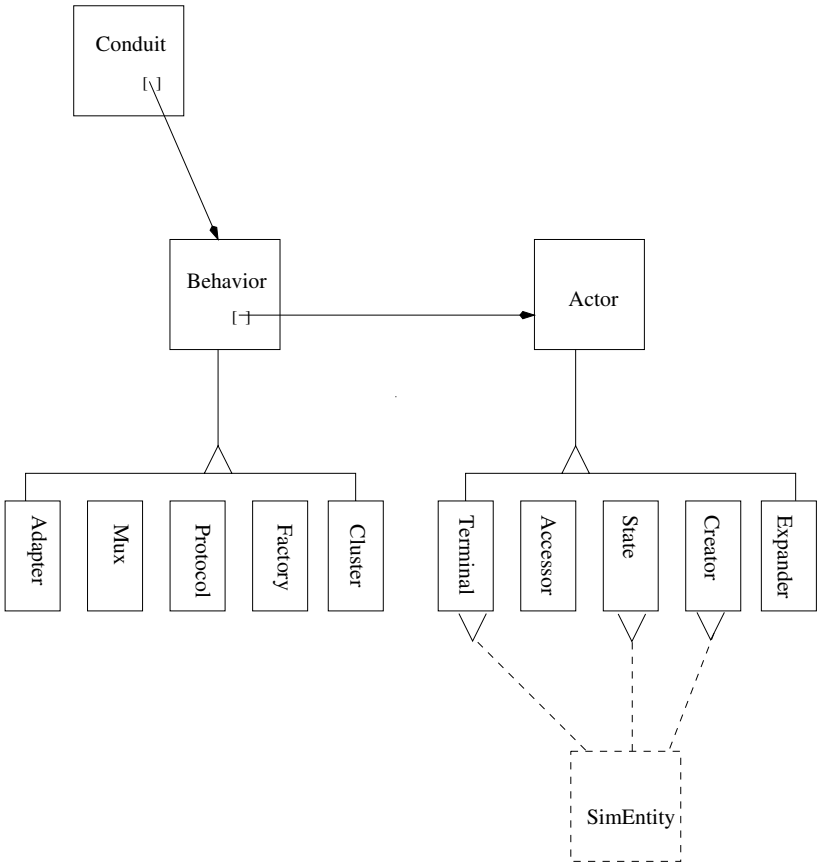

Figure 2. Conduit, Behavior and Actor.

There are five subclasses of the Actor class: Terminal, Accessor, State, Creator and Expander. These subclasses of the Actor correspond, respectively, to the Behavior's subclasses, Adapter, Mux, Protocol, Factory and Cluster. Figure 2 illustrates this relationship among Conduit, Behavior and Actor. The one-to-one correspondence among the instances of Conduits, Behaviors and Actors is rooted in the belief that a black-box framework is easier to use, more reusable and more easily extensible than the 'white-box' framework. [8] A white-box approach would define the five different types of Behaviors as subclass of the Conduit class, while the black-box approach lets the Conduit have a reference to these Behaviors. A common base class for these five is defined as the class Behavior and the Conduit class has a reference to a Behavior object. This is how one-to-one correspondence between Conduit and Behavior is achieved. Regarding the one-to-one correspondence between Behavior and Actor, the constructor of each subclass of Behavior only takes the appropriate corresponding subclass of the Actor

object as a parameter. At the Conduit level of abstraction, the basic unit (Conduit) is simply an object that can transport Visitors. At the Behavior level of abstraction, we identify semantic classes of Conduit operation. For example, a Mux operates by multiplexing Visitors in one direction and demultiplexing Visitors in another direction. A Terminal is a source/sink of Visitors. A Factory is a place where new Conduits are made. To fully specify a concrete Behavior, several details must be specified: for a Mux, 'how does a Mux determine where to demultiplex a Visitor to?'; for a Factory, 'what kind of Conduit should be made?', etc. This specification process could have been carried out in two ways: (i) by making certain methods abstract in the Behavior hierarchy and then requiring concrete versions of the Behaviors to implement these abstract methods; or (ii) by composition of the specification into the Behavior. We chose the latter approach and we call the specification objects Actors. If we consider CASiNO as a high-level language, the set of Behaviors are in essence the syntactic classes of the language, whereas the set of Actors are concrete words within the syntactic classes. We wanted to make the act of extending syntactic classes different from the action of implementing new concrete words within a syntactic classes. The latter is commonplace in designing a CASiNO application; the former constitutes extending the CASiNO framework.

Conduits receive and are given Visitors through their `Accept(Visitor *v)` method. The code illustrated in Example 2 sends the Visitor *v* to the Conduit *C*.

Example 2.

```
Visitor * v = new Visitor();
Conduit * C = new Conduit(``MyConduit'', _my_protocol);
C->Accept(v);
```

Visitors may be thought of as transient objects, usually representing communication messages or data flow. However, because Visitors are full C++ objects they are much more than just passive data packets or 'buffers'. A Visitor is a smart object, with the potential of having its own logic, that interacts with the logic of user-defined Actors. So at each Conduit, a dialog between the Visitor and the underlying Actor ensues and the outcome of this dialog determines the side effects within the state of the Visitor and the state of the Actor.

Precisely, the Visitor is first informed of the type of Conduit where it now finds itself situated. This is done by calling the appropriate one of the following `at()` methods of the Visitor:

```
void Visitor::at(Protocol * b, State    * a)}
void Visitor::at(Factory  * b, Creator  * a)}
void Visitor::at(Mux      * b, Accessor * a)}
void Visitor::at(Adapter  * b, Terminal * a)}
```

All of these methods may optionally be redefined within the application–derived Visitors, although these methods have reasonable default definitions in the base Visitor class. Within these `at()` methods, the Visitor determines how to proceed by evoking methods of its local actor, given by the second argument `a`. Typically, these actor methods require passing the identity of the Visitor itself as an argument, so the Actor also has the opportunity to respond to the identity of the Visitor.

Programmers who use CASiNO must derive all of their Visitors from the abstract base class Visitor. All Visitors must be allocated on the heap. Destruction of a Visitor takes place by calling its `Suicide()` method.

When making a Conduit, the programmer is required to specify its Behavior. As mentioned earlier, there are five types of Behaviors, each representing general semantics of a Conduits purpose and activity.

(1) Protocol Behavior—contains the logic for a finite state machine.
(2) Adapter Behavior—absorbs and/or generates new Visitors; interfaces with non-framework software.
(3) Mux Behavior—demultiplexes Visitors to (and multiplexes visitors from) a set of adjacent Conduits.
(4) Factory Behavior—creates and attaches new Conduits to neighboring muxes.
(5) Cluster Behavior—encapsulates a subnetwork of Conduits.

The programmer must fully specify a behavior at the time of construction with an appropriate concrete Actor object. There are, correspondingly, five types of Actors:

(1) a State Actor is required to specify a Protocol;
(2) a Terminal Actor is required to specify an Adapter;
(3) an Accessor Actor is required to specify a Mux;
(4) a Creator Actor is required to specify a Factory; and
(5) an Expander Actor is required to specify a Cluster.

We now describe each of these Behaviors and their associated Actor.

3.1. Protocol Behavior–State Actor

A communication protocol is often specified by a finite state machine (FSM). A Protocol Behavior implements a finite state machine using a specialized State Actor. When a Visitor arrives at a Protocol, it is handed to the State Actor. The State then queries the Visitor to dynamically determine what type of Visitor it is. Once the State determines the type of the Visitor, the State transforms the Visitor into the appropriate type casting and operates on the Visitor accordingly. This might trigger a state transition in the FSM. Additionally, the state transition itself could translate into a new Visitor being born and sent either to the Conduit on side A or side B. The State Actor is implemented, resembling the state pattern in [1]. The Behavior class keeps a protected Actor instance variable:

```
protected:
\\...
  Actor   * _actor;
\\...
```

so the Protocol class inherits the Actor instance variable. The State class is derived from the Actor class and subclasses of the State class, which correspond to the states of the FSM, can be defined. Each subclass can implement operations specific to its corresponding FSM state. Thus, the Protocol class can maintain an instance of a subclass of the State class.

We now give a more detailed account of the Protocol/State pattern. A Visitor arrives at a Conduit via the method

```
void Conduit::Accept(Visitor * v);
```

This evocation, in the CASiNO framework, results in calling the Visitor's method

```
v->at( Protocol* p, State *s )
```

passing in the current Conduits Protocol and State as arguments. If the Visitor has not redefined this `at()` method, then the base Visitor's method is called, which results in the State object's Handle method being called:

```
State* State::Handle(Visitor *v)
```

which is a pure virtual function of the State actor class and so must be defined in any user-defined State object. The programmer specifies the actions taken by the FSM in response to the arrival of Visitors in `Handle()`. This `Handle()` function must return a pointer to the new State object, i.e. the outcome of handling the Visitor. The Visitor then updates the Protocol's data structures to reflect the returned State; namely, the aforementioned Actor instance variable is updated to point to the new object representing the new FSM state.

The FSM data/logic is typically encoded in concrete State Actors. If the FSM is simple enough, the State's member function Handle may implement the logic directly. Alternatively, the visitor may be given to other singleton instances (of the State's subclasses) that implement the logic of the FSM. In the latter case, these singleton instances operate on the Visitor and modify the data State of the FSM.

3.2. Adaptor Behavior–Terminal Actor

The Adapter Behavior as its name indicates, converts or adapts the outside world to the Conduits and Visitors comprising the CASiNO application. Therefore, only its side A is connected to a Conduit. It is used typically for testing, where a Protocol stack is capped at both ends with two Adapters—one to inject Visitors and the other to sink them. The *Terminal* actor is used to configure the Adapter with a simple interface to allow Visitors to be injected and absorbed. As such, the Terminal class has a public function `Inject()` and a public pure virtual function `Absorb()`. `Inject()` calls the `Accept()` function, which is introduced in Example 2, to send a Visitor to the Conduit connected to side A of the Terminal. Figure 3 illustrates the use of a Conduit with an Adapter Behavior and a Terminal Actor, which is named 'Term'. Note that only side A of a 'Term' is connected to another Conduit. The architecture depicted is an intermediate phase in the implementation of the ATM's PNNI (Private Network-Network Interface) protocols in PRouST [5] over CASiNO. The schematic of Figure 3 shows the software architecture with inter-operation of the NodePeer (NP) protocol and the Hello protocol over a link prior to integration into the larger system.

3.3. Mux Behavior–Accessor Actor

In the real network, a packet often arrives at a point where there are many possible branches, perhaps going to different software or hardware modules. In such situations, the actual path is often determined by the contents of the packet. For example, an ATM cell flowing into an ATM switch through an input port could potentially depart from the switch through any of many different output ports and the output port of departure is determined by the Virtual Path Identifier (VPI) and Virtual Channel Identifier (VCI) fields of the cell's header. As another example, the IP protocol layer of a host interfaces both UDP (User

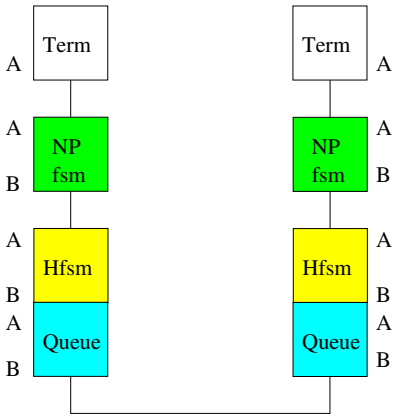

Figure 3. Adapter with a Terminal used to test the PNNI Hello and Node Peer FSMs.

Datagram Protocol) and TCP upper layers [26]. It sends the payload (data) of the received IP datagram to an appropriate upper layer protocol on the basis of 'Protocol' [26, p. 92] field of the IP header. Conversely, packet streams are often merged at some points. For example, both TCP and UDP packets are merged in the IP layer. For another example, ATM cells entering an ATM switch through different input ports may depart through an identical output port.

The Conduit with a Mux Behavior allows developers to implement multiplexing and demultiplexing. The demultiplexing occurs when a Visitor arriving on side A is routed to one of the several other Conduits on the basis of some aspect of the Visitor's contents, which we refer to as the Visitor's key or address. The Accessor Actor is responsible for examining the incoming Visitor's contents and deciding which Conduit the Visitor should be sent to. The Accessor Actor implements the Strategy pattern, where the Mux behavior is the context of the Strategy and the Accessor the Strategy [1]. The Accessor class shows the pure virtual function in its interface:

```
virtual Conduit * GetNextConduit(Visitor * v) = 0;
```

which returns the pointer to the next Conduit. The programmer must derive a subclass of Accessor and implement this function. To determine the next Conduit for the Visitor to visit, there must be a map that associates the Visitor's keys to Conduit pointers. When the programmer designs a particular Accessor (a class derived from the Accessor base class), the programmer must include in that derived class a data structure implementing the map. Then, the implementation of `GetNextConduit(Visitor * v)` can look up the map to determine the next Conduit. Figure 4 shows two Conduits with Mux Behavior. We represent a Mux Conduit by a trapezoid. In Figure 4, side B of both Muxes are attached to a common Factory Conduit. In the pictorial representation of a CASiNO application, the line between the long side of the Mux diagram (trapezoid) and its neighboring Conduit means that the neighboring Conduit is known to the Accessor of the Mux. That is, the neighboring Conduit is included in the Accessor's map. In Figure 4, Conduits named Call1, Call2 and Call3 are listed in the map of the

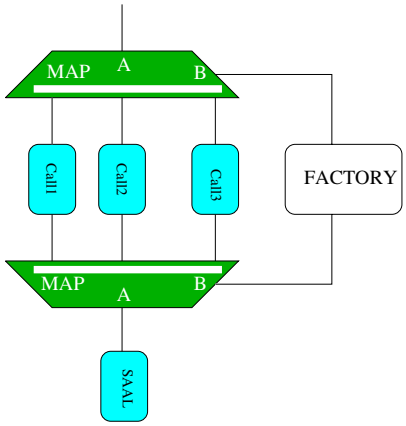

Figure 4. Common usage of two Muxes and a Factory.

Accessor belonging to the Mux Conduit above. They are also listed in the map of the Accessor belonging to the Mux Conduit below.

Dialogues between the Accessor and the Visitor take place before the Visitor's next Conduit is determined. A Visitor arriving from any Conduit in the Accessor map is automatically sent to the Conduit on side A—this is the multiplexing effect (see Figure 5). If a Visitor enters the Mux Conduit from side A, the Accessor's method `GetNextConduit(Visitor * v)` determines the next Conduit, as illustrated in Figure 5. If a mapping does not include a particular Visitor's key, then the Visitor is sent to the Conduit attached to its side B. Side B of a Mux is attached to the default Conduit. Thus, an arriving Visitor whose key maps into a NULL Conduit is sent to the Conduit on side B, usually a Conduit with a Factory behavior, as illustrated in Figure 6. The Factory Behavior creates a new Conduit on the basis of the Visitor's contents and allows the Visitor to be routed to this new Conduit.

3.4. Factory Behavior-Creator Actor

The Factory Conduit allows dynamic instantiation of new Conduits at Muxes. Thus, a Factory Behavior is always found with Muxes (see Figure 4). A Conduit configured with a Factory Behavior is connected to side B of a Conduit configured with a Mux Behavior. Upon receipt of a Visitor, the Factory may make a new Conduit. If it decides to do this, the Factory makes the newly instantiated Conduits known to the Accessor of the Mux by using the Accessor's `Add(Conduit *c, Visitor * v)` method. This Add method is a pure virtual function and the implementor must define it when designing the concrete Accessor associated with the Mux. The Add method generates new keys and new entries in the Accessor's map for the newly added Conduit. In this manner, Visitors can be sent to a Mux to install new Conduits dynamically, provided that a Factory is attached to the Mux. The Factory Behavior is specified with a Creator Actor. The Creator Actor instantiates a new Conduit object on the basis of the identity and contents of the incoming Visitor. After the installation (instantiation and connection with

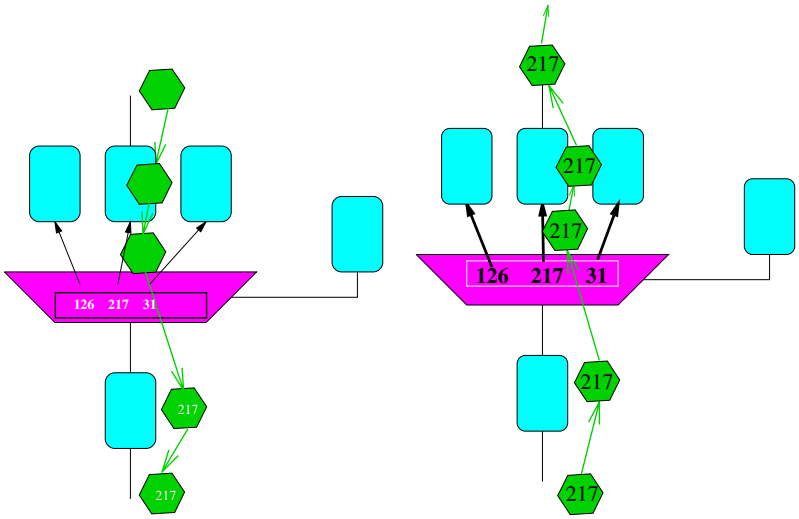

Figure 5. During multiplexing (left) a Visitor gets stamped with the key at the Protocol; during demultiplexing (right) a Visitor is sent to Protocol with a matching key.

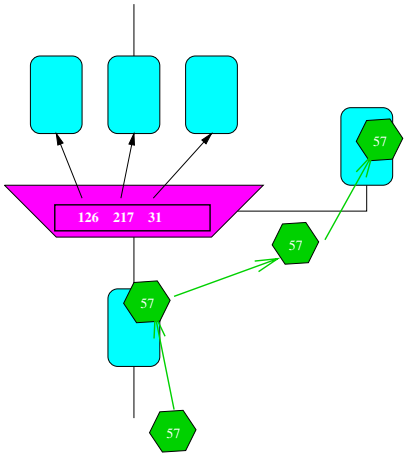

Figure 6. No matching key in the Accessor, Visitor goes to Factory.

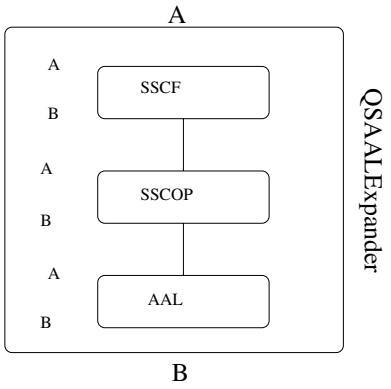

Figure 7. Cluster Behavior.

the Mux) of the newly created Conduit object, the incoming Visitor, which prompted the installation, is handed to the freshly instantiated Conduit by the Factory. The Factory does not stamp the Visitor, so the Visitor's last seen Conduit is the Mux that handed it to the Factory. Now the Visitor has an entry in the Accessor's map and can continue its flow through the Conduits. If both sides of a Factory are connected to Conduits with a Mux Behavior as in Figure 4, the newly created Conduit is made known to both Muxes.

3.5. Cluster Behavior–Expander Actor

In addition to the four primitive Behaviors of Protocol, Factory, Mux and Adapter, the CASiNO has generic extensibility via the Cluster Behavior. Just as the primitive behaviors mentioned above are configured with a particular Actor (i.e. a concrete State, Creator, Accessor or Terminal object), the programmer can fully specify a Cluster's behavior by configuring it with a concrete Expander Actor. An Expander may be thought of as a black-box abstraction of a network of Conduits. In using the Cluster/Expander mechanism, programmers may design higher-level building blocks from the primitive framework components. Such customized higher-level components continue to have all the privileges of primitive CASiNO classes. Clusters/Expanders can thus be thought of as a mechanism to abstract away groups of Conduits into a single Conduit.

For example, in implementing the ATM User Network Interface (UNI), a programmer may decide that several spatially proximate Conduits need to be instantiated every time a new call arrives. Such Conduits are candidates for being grouped together into a single Expander called, for example, UNI_Call. That way, for each call arrival the program can instantiate a single Conduit with a Cluster Behavior specified by this UNI_Call Expander.

In order to specify an Expander, the programmer must declare the network of constituting sub-Conduits that are being abstracted away as a unit. We now discuss an example of a subclass derived from Expander, which we refer to as 'QSAALExpander'. Figure 7 pictorially illustrates

this subclass. QSAALExpander is a subclass used to implement the signaling AAL layer of ATM networks. Three Conduits named 'AAL', 'SSCOP', 'SSCF' constitute the 'QSAALExpander' class. The following is an abridged version of QSAALExpander's interface:

```
  class QSAALExpander : public Expander {
  public:
    QSAALExpander();
    virtual Conduit *GetAHalf(void) const;
    virtual Conduit *GetBHalf(void )const;
\\ ....
  protected:
    virtual ~QSAALExpander(void);
\\ ....
  };
```

The member variables _sscf, _sscop and _aal are pointers to the three constituting Conduits. As illustrated in this code, the programmer designing a particular Expander must include such member variables in its declaration. The constituting Conduits are instantiated when the Expander is instantiated. The invocation of the Expander's constructor instantiates them. We present again an abridged version of the QSAALExpander's constructor:

```
  QSAALExpander::QSAALExpander()
  {
    SSCFconn  * sscf_fsm  = new SSCFconn();
    Protocol  * sscf_protocol = new Protocol(sscf_fsm);
    _sscf = new Conduit("SSCF", sscf_protocol);

    SSCOPconn * sscop_fsm = new SSCOPconn();
    Protocol  * sscop_protocol = new Protocol(sscop_fsm);
    _sscop = new Conduit("SSCOP", sscop_protocol);

    SSCOPaal  * aal_fsm = new fore_SSCOPaal();
    Protocol  * aal_proto = new Protocol(aal_fsm);
    _aal = new Conduit("AAL",aal_proto);

    Join(B_half(_sscf), A_half(_sscop));
    Join(B_half(_sscop), A_half(_aal));

    DefinitionComplete();
}
```

Two invocations of the Join function in this code connects side B of SSCF to side A of SSCOP and side B of SSCOP to side A of AAL, which is illustrated in Figure 7. The constructor of the Expander implements the network structure of the constituting Conduits. In addition to the structure of the constituting Conduits, the programmer must specify side A and side B of the Expander object. That is, the programmer must indicate which of the lower-level components will be considered the 'A'

side of the new Expander and which of the lower-level components will be considered its 'B' side—a Cluster can only have a one unique conduit as its A interface and one unique conduit as its B interface. The programmer specifies those through the Expander's pure virtual functions

```
virtual Conduit *GetAHalf(void) const;
virtual Conduit *GetBHalf(void )const;
```

In the case of QSAALExpander, we have

```
Conduit * QSAALExpander::GetAHalf(void) const { return A_half(_sscf); }
Conduit * QSAALExpander::GetBHalf(void) const { return B_half(_aal);  }
```

It should be pointed out that the Cluster/Expander paradigm is purely a grouping mechanism. Visitors flowing through a network never sense that they have entered a Conduit of Cluster Behavior. In fact, Visitors always flow between primitive Conduits at the lowest level of the abstraction hierarchy and are not aware of abstraction structures imposed by the programmer.

4. THE REACTOR KERNEL

CASiNO can be used to implement both live network protocols as well as their simulations. Both cases require precise coordination of program execution relative to some notion of time. In the case of network simulation, however, the notion of time is quite different because the time within the simulation need not remain in lockstep with the real wall clock time. The Reactor Kernel can switch between these paradigms transparently and user programs are not affected in any way (in fact may be oblivious to) whether they are operating in simulated or live mode.

The Reactor Kernel library consists of several classes: Kernel, SimEntity, Handler and SimEvent. Throughout the simulation there is only one Kernel object (belonging to the Kernel class) instantiated. A reference to this singleton is returned by the global function

```
Kernel& theKernel()
```

The Kernel object is responsible for coordinating the registration and execution of events at discrete points in time. The application programmer has the ability to control the compression or expansion of time by setting the 'Speed' parameter of the Kernel. This is accomplished by the method

```
void  SetSpeed(Speed newspeed);
```

Speed is an enumeration defined in the Kernel class; some of the more useful speeds are `REAL_TIME`, `DOUBLE_SPEED`, `HALF_SPEED` and `SIM_TIME`. Operating the kernel at `SIM_TIME` speed permits discontinuous jumps in the kernels notion of time during stretches where no event is registered to take place. This allows the simulation to progress as quickly as possible relative to the wall clock. The Kernel represents time using the KernelTime class, which simply encapsulates the system time variable and provides useful operations and security.

The Kernel object maintains a queue of scheduled events, ordered by their expiration times. At each expiry the Kernel sends a message to the objects, i.e. invokes a user-redefinable operation of the relevant

object that needs to be notified regarding the event. Those objects can then take proper actions in response to the events occurrence, after which they return the program control back to the Kernel. As the Kernel object prompts objects to take certain actions, these objects may also alter the Kernel state; for example, they may schedule or cancel other future events. Such actions are referred to as registration and deregistration of events with the Kernel. Upon a registration or deregistration, the Kernel updates its event queue appropriately and then continues its regular operations.

An object that is capable of requesting the service of the Kernel is referred to as (i.e. must be derived from the class) SimEntity. SimEntity provides a standardized and secure interface through which the Kernel is accessed, both for management of timers and for I/O. Typically, a SimEntity object instantiates a derived Handler of one of three types, TimerHandler, InputHandler or OutputHandler, and uses its (SimEntity's) protected Register() methods to notify the Kernel that it requests the registration of that particular Handler. The Handler class is an abstract base class that defines a uniform interface for callbacks, which are the methods to be evoked by the Kernel when certain conditions are satisfied. The TimerHandler is a time-based callback. A TimerHandler encapsulates an expiration time. When it is registered, an event is scheduled at the expiration time in the Kernel object's queue. When that time is reached, the Kernel will call that TimerHandler's Callback method. The `Callback()` method must be defined by the programmer to fully specify a handler object. Conceptually, the most essential elements of the TimerHandler are encapsulated expiration time and the Callback function.

InputHandler and OutputHandler encapsulate a descriptor (file descriptor, socket descriptor, etc.). InputHandlers and OutputHandlers that are registered with the Kernel are called back when there is either data to be read from the device represented by the descriptor (in the case of the InputHandler) or when the device is available for writing (in the case of the OutputHandler). The Callback function code prescribes actions to be taken by the Input/OutputHandler in those circumstances. Again, the `Callback()` method must be defined by the programmer to fully specify a handler object. InputHandlers and OutputHandlers are the interface through which the framework application exchanges data with the outside world. InputHandlers and OutputHandlers can respond to any file or device that can be represented by a descriptor (network sockets, files, named pipes, etc.). Conceptually, the most essential elements of the InputHandler or OutputHandler are the encapsulated file descriptor and the Callback function.

In addition to the ability to register Handlers, the SimEntity class provides its instances with the ability to interrupt themselves or other SimEntity instances. When an interrupt needs to be scheduled, a SimEvent object is instantiated. The SimEvent object encapsulates the reference to the target SimEntity object and such information must be provided to the constructor at the time of the SimEvent object's instantiation. The operation that is requesting the interruption then evokes the SimEntity's member function (operation)

```
Deliver(SimEvent * e, double time)
```

This member function eventually results in the target SimEntity's `Interrupt(SimEvent* v)` member function being called after the specified time given as the second argument, 'time', which is expressed in seconds. The interruption time is relative to the time that the method `Deliver()` is called. (In fact, the `Deliver()` member function instantiates a special TimerHandler object and registers it to the Kernel object. The Kernel object calls the special TimerHandler object's callback function

at the proper time and the callback function calls the Destination SimEntity object's Interrupt function.) A programmer must write the Interrupt method when defining a class derived from SimEntity class.

Among the variables encapsulated by the SimEvent class that are important are the source SimEntity, Destination SimEntity and the integer code. This set of information is passed to the constructor when the SimEvent object is created. The integer code contained in the SimEvent is used by the receiving SimEntity to differentiate among the derived types of SimEvents. This allows the recipient to respond to SimEvents containing a different code number with a different action. Two SimEntity objects can have a dialogue with each other by sending the same SimEntity object back and forth through the Deliver operation. The destination SimEntity can interrupt in response to the origin SimEntity by using the ReturnToSender operation. This function takes a pointer to a SimEvent object and time to deliver, just as the case of the Deliver operation. The destination SimEntity can also alter the code of a received SimEvent before sending it back to the origin SimEntity object.

4.1. Summary of the Reactor Kernel

- Kernel. Responsible for keeping track of an application's Timers, I/O Handlers and SimEvents that are pending delivery between SimEntities.
- SimEntity. A client of the Kernel. A concrete SimEntity must specify the Interrupt(), method which will be called when a SimEvent arrives. A SimEntity may register Timers and I/O Handlers and send SimEvents to other SimEntities.
- Handler. A contract with the Kernel. A concrete Handler must specify a Callback() method, which the Kernel will call when a Timer expires or I/O occurs.
- SimEvent. A 'smart interrupt' that the Kernel delivers directly between SimEntities.

4.2. Mechanics of the Kernel operation

The Kernel keeps a priority queue that lists all the registered TimerHandler objects, ordered according to their expiration times. The execution of the Kernel consists of a two-phase loop as follows.

- Phase I. The Kernel checks the priority queue. If it is time for the next timer to expire, the Kernel removes that TimerHandler from the queue and executes its Callback method.
- Phase II. If the expiration time of the next event has not yet been reached, then the Kernel checks the devices associated with registered InputHandlers and OutputHandlers. For each of the descriptors that are ready for reading, the Kernel calls the associated InputHandler's Callback method. For those descriptors that are ready for writing, the Kernel calls the associated OutputHandler's Callback method.

5. INTEGRATION OF DATA-FLOW AND REACTOR KERNEL

Three of the Actors, Terminal, State and Creator, are derived from the SimEntity class as well as the Actor class, as illustrated in Figure 2. These three classes thus bridge the Data-Flow Architecture and the Reactor Kernel. Conduits that are specified by instances of Terminal, State or Creator thus have access to both a SimEntity-based model of interaction and a Visitor-based model of interaction.

Table I. Summary and comparison of Reactor Kernel and Data-Flow Architecture.

Name	Reactor Kernel	Data-Flow Architecture
Deals with	Program flow through time	Program flow through space
Transient atoms	SimEvent, system event	Visitor -na-
Location of atom	SimEntity, Handler	Conduit/Behavior/Actor -na-
Location's response	SimEntity::Interrupt Handler::Callback	Actor specification -na-
Atom's response	-na-	Visitor specification

Visitors may be thought of as 'smart data packets' that travel through a network of software Conduits, while SimEvent are 'smart interrupts' delivered directly between Conduits.

Two Conduits wishing to communicate should use the Visitor model in situations where: (a) the connectivity of intermediate Conduits is static (or at least predictable); and (b) intermediate Conduits need to augment or witness the communication between two Conduits. Visitor-based communication is ill-suited for situations where stringent temporal guarantees are needed for when the message will arrive at the destination Conduit. To clarify, we note that achieving low-latency with Visitor-based communication requires all intermediate Conduits to not impede or delay the passage of the Visitor. This is not something CASiNO can itself guarantee, since the behavior of Conduits is specified by the programmer. In short, if the programmer is certain that intermediate Conduits will not impede or delay the Visitor, then low-latency communication between distant Conduits can be achieved. This certainty, however, is most likely to manifest in programs where the interconnectivity of intermediate Conduits is predictable to the programmer. Two Conduits should consider using SimEvents for communication when: (a) the connectivity of intermediate Conduits is dynamic and cannot be easily predicted; or (b) stringent temporal guarantees are needed for when the message will arrive at the destination Conduit. Table I summarizes comparisons of the substructures of the Reactor Kernel and the Data-Flow Architecture.

6. AN EXAMPLE OF A CASiNO APPLICATION

The CASiNO application in Appendix A implements a simple slot machine. There are three wheels in the slot machine. Each wheel has four symbols on it: a banana, a cherry, a ring and a bar. The simulation begins with a bank of 10 000 coins for the house and 100 coins for the player. There is a payoff whenever the symbols on all three wheels match.

The application shows how to make Conduits that behave like Adapters, Factories, Muxes and Protocols. It illustrates the creation, motion through Conduits and deletion of Visitors. The application writes a summary of the lives of all of its Conduits and Visitors in the file 'casino.vis' so that the

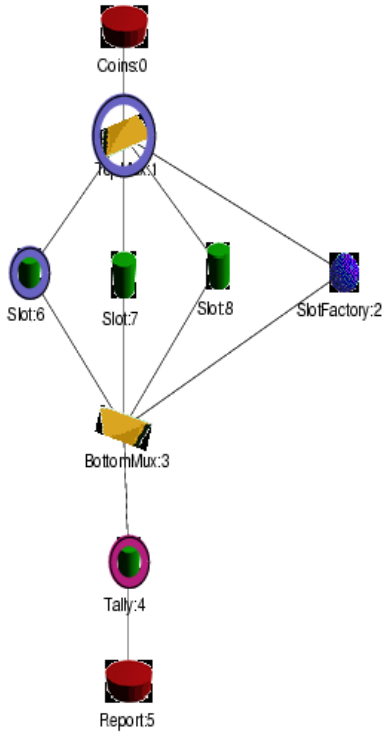

Figure 8. Visualizing the CASiNO slot machine example.

simulation can be animated by one of the CASiNO visualization tools, which is described in Section 7. Visualization of the slot machine application is illustrated in Figure 8. The following sections provide expository comments on the example code. We intend to illustrate how the Kernel and the Data-Flow Architecture interact and how the CASiNO framework is used to create a program.

6.1. Program requirements

The goal of the example programming project is to develop a simple simulator that simulates the operation of a slot machine. The operation has the following procedures.

(1) A coin is dropped.
(2) In response to the coin dropping, the slot machine shows one of the values *lemon*, *cherry*, *ring* or *bar* in each of three slots of the machine.
(3) On the basis of the values shown in three slots, the slot machine computes the prize.

6.2. Object-oriented analysis

We now demonstrate the process of converting the program requirements into the object-oriented program architecture by using CASiNO and simple physical intuitions. First, the slot machine player drops a coin into the machine. Since the slot machine player is an input interface point between the external world and the software system under construction, the player is best implemented as a Terminal Conduit. We name this Terminal Conduit 'Coins' because from the point of view of the software, the slot machine player is nothing more than a source of coins (see the top of Figure 8).

Each coin in turn can be viewed as a message passing to the slot machine, signaling play. It is thus quite natural to model the coin using a Visitor. The program listed in Appendix A defines a derived class `PlayVisitor` to model the coin. When a coin is received by the slot machine it must cause each of the three slots to display new values. From this branching structure, simple physical intuition calls for a Mux conduit to be adjoined to the 'Coins' Terminal, which in turn fans out to three Conduits representing the three slots. This Mux is depicted Figure 8, where it is referred to it as the 'TopMux'. Since each slot has a persistent state, it can be implemented using a State Conduit. These are depicted in Figure 8, where they are each named 'Slot'. Each of these State Conduits will determine a value from the set *lemon*, *cherry*, *ring*, *bar*.

Now, since the information about the values of the slots needs to be aggregated in order to compute the prize, we consider placing another Mux Conduit to merge information about the three slots. This Mux is depicted in Figure 8, where it is referred to it as the 'BottomMux'. The state of each slot needs to be conveyed to a common point where the merging can take place; this state information could be carried by another type of Visitor. The program defines a `SlotVisitor` class for this purpose. The BottomMux forwards all SlotVisitors arriving from above to another Conduit which can compute the prize and presents the outcome. This Conduit is named the 'Tally' in Figure 8. A Visitor arriving from any Conduit in the Accessor map is automatically sent by internals of CASiNO to the Conduit on side A, which in the case of Figure 8 is the Tally object. The Tally object then computes the prize on the basis of the values of three slots carried in SlotVisitors.

The information about the prize determined by Tally must be conveyed to the displayer of the slot machine. This information could be embodied in a specialized Visitor, which we call a ReportVisitor. Since the displayer is an output interface point between the software system under construction and the external world, it is best implemented as a Terminal Conduit; this object is named 'Report' in Figure 8. According to the necessity of data flow between Tally and Report, the Conduits are joined. This completes the operational architecture of the slot machine. Note that for simplicity, in this exposition we have omitted the SlotFactory Conduit in Figure 8 since it is only active when the slot machine is first instantiated and not during the subsequent normal operation.

6.3. Expository comments

According to the analysis presented in Section 6.2, the programmer of the slot machine needs to implement a few specialized Conduit classes, namely the 'Coins' Terminal, the 'Slot' State, the 'Tally' State, the 'Report' Terminal and Accessors for the 'TopMux' and 'BottomMux'. The programmer must also define the specialized Visitor classes PlayVisitor, SlotVisitor and ReportVisitor. Having done this, CASiNO makes the `main()` function of the slotmachine program extremely simple: it simply

instantiates the objects and joins them according to the object-oriented analysis, then the Kernel is set into motion and as a consequence the program runs.

In this section, we provide some more details on how each of the specialized Conduits and Visitors operate. In doing so, we hope to provide the readers with more solid understanding of the use of CASiNO, and also a better picture of how the internals of CASiNO operate to run the program.

6.3.1. Initializing the software architecture

The main program is initiated by obtaining a reference to the Kernel object in the line

```
Kernel & theWorld = theKernel();
```

The constant `SIM\_SPEED` is used

```
theWorld.SetSpeed(Kernel::SIM_SPEED);
```

to specify that the speed of the simulation should be 'as fast as possible'. Then the program instantiates the specialized Conduits of the application.

```
Conduit *coinC      = new Conduit("Coins", new CoinTerminal(6));
Conduit *topMuxC    = new Conduit("TopMux", new SlotAccessor(3));
Conduit *factoryC   = new Conduit("SlotFactory", new SlotCreator());
Conduit *bottomMuxC = new Conduit("BottomMux", new SlotAccessor(3));
Conduit *tallyC     = new Conduit("Tally", new TallyState(3));
Conduit *reportC    = new Conduit("Report", new ReportTerminal(10000, 100));
```

The Conduits are then 'joined' into the network illustrated in Figure 8. Finally, calling

```
theWorld.Run();
```

starts running the Kernel in motion, as described in Section 4.2.

As part of initializing the architecture, the three 'Slot' State Conduits which implement the machine's three slots must be instantiated and made known to TopMux and BottomMux objects. The 'Coins' Terminal object makes this happen using the semantics of the Muxes and Factories. Specifically, the example code in Appendix A is designed so that the 'Coins' Terminal injects three SlotVisitors into the machine immediately upon instantiation. Since initially no 'Slot' States are present, these three SlotVisitors are routed by the TopMux to the SlotFactory. The SlotFactory instantiates a 'Slot' State Conduit for each of these Visitors and installs these Slot States between the TopMux and the BottomMux. This completes the initialization of the software architecture.

6.3.2. Operation of the software

During normal operation, the 'Coins' Terminal uses the `Deliver()` method of SimEntities to send itself a SimEvent every `_interval` seconds. In response to the arrival of this event (i.e. in its `Interrupt()` method) the 'Coins' Terminal sends a newly created PlayVisitor by injecting it downwards, into the TopMux object, signaling the beginning of another play. (The 'Coins' Terminal

injects the PlayVisitor by calling the `Inject()` function. The `Inject()` function is the Terminal class' operation that passes the Visitor object referenced by the argument to the neighboring Conduit.)

Whenever a PlayVisitor finds itself in a Mux object (i.e. when its `at(Mux * b, Accessor * a)` method is evoked), the PlayVisitor broadcasts itself to all attached Conduits using the Accessor's `Broadcast` method. In our application architecture, this causes PlayVisitors to be replicated and to enter each of the three Slot States from above.

When a Slot State receives a PlayVisitor, it chooses a random number from 1 to 4, destroys the PlayVisitor and then sends information about its randomly chosen state (i.e. whether it is *lemon, cherry*, *ring* or *bar*) downwards using a newly created SlotVisitor.

SlotVisitors are received by the BottomMux and forwarded downwards, as part of normal multiplexing operations. In our application, this results in sending the SlotVisitors into the Tally State Conduit. The Tally State processes the contents of the SlotVisitors and maintains information about the cumulative state of all Slots. It kills incoming SlotVisitors using their `Suicide()` method, once it has incorporated their information. Every time the Tally object has received SlotVisitors from all three Slots, it recomputes whether a prize is to be awarded. The Tally State puts its verdict into a newly made ReportVisitor, which is then sent downwards into the 'Report' Terminal Conduit.

The 'Report' Terminal examines the ReportVisitor, displays the verdict on the screen, and then kills the ReportVisitor.

7. VISUALIZATION OF CASiNO APPLICATIONS

CASiNO includes NV, a tool developed independently of other network visualization tools, for visualizing CASiNO simulations. NV is a general animation tool, with two kinds of objects—fixed objects (Conduit in CASiNO) and moving objects (Visitor in CASiNO). The moving objects start their journey in the fixed object that instantiated them and jump from one fixed object to another until they get destroyed. They arrive at their next fixed object when the Accept(Visitor v) method is called on the fixed object. To be able to animate such a situation, we need a list of all fixed objects and their respective picture icons as well as the links between them to draw the possible pathways. NV internally keeps this information in a graph abstract data structure. In addition, we also need to know the creation, arrival and destruction of the moving objects. This is accomplished in CASiNO by having each fixed object write a line in a log file in their constructor and destructor to signal their birth and death. The syntax used in the log file is not designed for convenience of human reading, but rather for the efficient encoding to be read by the NV module. For example, the line in

```
!birth Umux:1 !Mux
```

the log file indicates the creation of a fixed object representing a multiplexer. Similarly, the line

```
!connected Umux:1 2 !to sockFac:2 1
```

indicates to link the multiplexer Umux to the socket factory sockFac. In a similar fashion, moving objects births are recorded with the additional information specifying where they were instantiated or which fixed object instantiated them. For example, to indicate a birth of a Visitor named hello in a Conduit named Top, the following log statement is used:

```
!birth hello:5 !Visitor !in Top:0
```

Finally, the movement of Visitors from Conduit to Conduit is recorded by the log statement

```
!received hello:5 !in Lmux:3
```

Notice here that NV has to remember the previous position of the Visitor in the graph of conduits so that it can erase its presence there and show it at the new position. NV displays the presence of Visitors at Conduits using concentric circles of different colors. For example if NV is used to animate the flow of packets in a network, congestion will be shown as multiple concentric circles at the queues and can be easily detected. NV allows the animation to proceed continuously or in a step-by-step fashion. In the step-by-step fashion display, NV updates its display in response to human commands by reading one statement of the log file at a time. NV also permits backward animation or rewinding.

Finally, Figure 8 shows a snapshot of the NV's rendering of the slot machine example of Section 6. The terminal at the top, 'Coins:0', is where the 'Coins' originate as PlayVisitors. A PlayVisitor in the mux 'TopMux:1' is waiting to be dispatched to one of the 'Slot' Protocols beneath it. There is a PlayVisitor in 'Slot:6'. A SlotVisitor has already passed through the Mux 'BottomMux:3' and into the Protocol 'Tally:4'. When Tally:4 has collected one SlotVisitor from each of the three Slot Protocols, it will decide what the payoff will be and inform the terminal 'Report:5' which will print the result.

8. CONCLUSIONS

The modular coarse-grained dataflow architecture of CASiNO, together with its reactor kernel for asynchronous I/O, timers and interrupts, make it an extremely useful framework for implementing communication protocol stacks and network simulators. Indeed, CASiNO has been used with great success as the foundation for two major development projects at the Center for Computational Sciences at the Naval Research Laboratory as follows.

- Signalling Entity for ATM Networks (SEAN), which implements the host native ATM protocol stack and the ATM User Network Interface, conforming 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.
- PNNI Routing and Simulation Toolkit (PRouST), which is a faithful and complete implementation of version 1.0 of the PNNI routing protocol. PRouST can be used to simulate large networks of ATM switches, as well as for software emulation of a switch protocol stack.

The development efforts for SEAN and PRouST have been a proving ground for CASiNO. Both of these projects have involved extensive software development by a teams of programmers over several years. CASiNO has provided a scalable and consistent methodology by which to modularize the components of protocols and distribute development efforts in terms of both time and people. A PRouST simulation of an ATM network, for example, may typically involve 10–100 000 Conduits and the number of visitors throughout the lifetime of the program's execution is in the millions. The structure imposed by CASiNO's paradigms has made the software easy to design, develop, maintain and describe. Indeed, the successes of the PRouST and SEAN initiative have demonstrated the robustness, scalability and richness of CASiNO and its viability as a framework for the development of network protocols. The software documentation of CASiNO is available at [27].

APPENDIX A. EXAMPLE: SLOT MACHINE

```
/* Demonstrates some features of CASiNO.  Simulates a slot machine.  */

// A PlayVisitor is an indication to play the slot machine once.
//
class PlayVisitor : public Visitor {
public:
  PlayVisitor(void);
  virtual ~PlayVisitor(void);

  // PlayVisitors broadcast themselves to an arbitrary number of slots at muxes.
  virtual void at(Mux *m, Accessor *a) { m->Broadcast(this); }
};

// A SlotVisitor carries the current state of a slot.
// The state is one of {lemon, cherry, ring, bar}.
// When a SlotVisitor enters a SlotCreator, the SlotCreator makes
// a new SlotState and inserts it between the top mux and the bottom mux.
//
class SlotVisitor : public Visitor {
public:
  enum SlotValue {no_value, lemon, cherry, ring, bar};

  SlotVisitor(int slot, SlotValue value=no_value) : _slot(slotl), _value(value) { }
  virtual ~SlotVisitor(void);

  int            _slot;
  SlotValue      _value;
};

// The ReportVisitor carries the states of all of the slots in the
// slot machine and the payoff for the combination of slots.
//
class ReportVisitor : public Visitor {
public:
  ReportVisitor(int num_values, SlotVisitor::SlotValue *value, int payoff)
    _num_values(num_values), _value(value), _payoff(payoff) {}
  virtual ~ReportVisitor(void);

  int Payoff(void) { return _payoff; };
  void PrintValues(void) {
    for (int i = 0; i < _num_values; i++) {
      switch (_value[i]) {
      case SlotVisitor::lemon:  cout << "lemon\t";  break;
      case SlotVisitor::cherry: cout << "cherry\t"; break;
      case SlotVisitor::ring:   cout << "ring\t";   break;
      case SlotVisitor::bar:    cout << "bar\t";    break;
      default: break;
      }
    }
  };
```

```
private:
  int                     _payoff, _num_values;
  SlotVisitor::SlotValue *_value;
};

// CoinTerminal sits at the top of the slot machine.  It initially creates all
// of the slots by sending 3 SlotVisitors into the machine.  Then, every interval
// seconds, it sends a PlayVisitor into the machine to start another game.
//
class CoinTerminal : public Terminal {
public:
  CoinTerminal(double game_interval) : _interval(game_interval),
    _slots_initialized(false) {
    _playEvent = new SimEvent(this, this);
    Deliver(_playEvent, _interval);
  };
  virtual ~CoinTerminal(void);

  // Every Terminal defines this method to handle SimEvents,
  // and we expect an event every _interval seconds in the CoinTerminal.
  virtual void Interrupt(SimEvent *s) {
    if (!_slots_initialized) {
      SlotVisitor *s = new SlotVisitor(0); Inject(s);
                       s = new SlotVisitor(1); Inject(s);
                       s = new SlotVisitor(2); Inject(s);
      _slots_initialized = true;
    }

    PlayVisitor *p = new PlayVisitor(); Inject(p);
    Deliver(_playEvent, _interval);
  };
private:
  SimEvent *_playEvent;
  double    _interval;
  bool      _slots_initialized;
};

// SlotState represents one of the wheels in the slot machine.  When
// it receives a PlayVisitor, the SlotState chooses a new face at
// random from the range of SlotVisitor::SlotValue (lemon..bar).  Then
// it sends a summary of its state in a SlotVisitor to the TallyState
// below the bottom mux. Each SlotState sits between the top mux and
// the bottom mux.
//
class SlotState : public State {
public:
  SlotState(int slot) : _slot(slot) {
    _value     = (SlotVisitor::SlotValue)(rand() % 4) + 1;
  };
  virtual ~SlotState(void);
  // Every State has one of these for handling Visitors.
```

```
  State *Handle(Visitor *v) {
    _value = (SlotVisitor::SlotValue)(rand() % 4) + 1;
    SlotVisitor *s = new SlotVisitor(_slot, _value);
    PassVisitorToB(s);
    v->Suicide();
    return this;
  };

private:
  int                    _slot;
  SlotVisitor::SlotValue _value;
};

// TallyState collects SlotVisitors from all of the SlotStates.  When
// it has one from every SlotState, it determines the payoff for that
// combination of states and sends the answer in a ReportVisitor to
// the ReportTerminal at the bottom of the slot machine. The
// TallyState sits between the bottom mux and the ReportTerminal.
//
class TallyState : public State {
public:
  TallyState(int slots) : _slots(slots) {
    _payoff = new int[5]
    for (int i=0;i<=4;i++) { _payoff[i] = (int) pow(2.0,i); }
    InitSlot();
  };
  virtual ~TallyState(void);

  // We expect only SlotVisitors here.
  virtual State * Handle(Visitor *v) {
    bool all_filled = true;
    bool all_identical = true;

    SlotVisitor *s   = (SlotVisitor *)v;
    _slot[ s->_slot ] = s->_value;

    // See if all slots have a value
    for (int i=0; i<_slots; i++) {
      all_filled    = all_filled    && (_slot[i] != SlotVisitor::no_value);
      all_identical = all_identical && (_slot[0] == _slot[i]);
    }

    if (all_filled) {
      int payoff = 0;
      if (all_identical) payoff = _payoff[ (int)_slot[0] ];

      // Give _slot to r then make a new _slot array.
      ReportVisitor *r = new ReportVisitor(_slots, _slot, payoff);
      _slot = 0;
      InitSlot();

      PassVisitorToB(r);
```

```
    }
    v->Suicide();
    return this;
  };

private:
  void InitSlot(void) {
    _slot = new SlotVisitor::SlotValue[_slots];
    for (int i=0; i<_slots; i++) _slot[i] = SlotVisitor::no_value;
  }

  int   _slots, *_payoff;  // slots, reward matching sets
  SlotVisitor::SlotValue *_slot;  // report from each slot;give to ReportVisitor.
};

// Print the result of a game.
// The ReportTerminal sits at the bottom of the slot machine.
//
class ReportTerminal : public Terminal {
public:
  ReportTerminal(int bank, int player) : _bank(bank), _player(player) { }
  virtual ~ReportTerminal(void);

  // A Terminal defines this method for handling Visitors,
  virtual void Absorb(Visitor *v) {
    ReportVisitor *r = (ReportVisitor *)v;
    int payoff = r->Payoff();

    if (payoff > 0) { _bank -= payoff; _player += payoff; }
    else            { _bank++; _player--; }

    r->PrintValues();
    cout << "\tplayer: " << _player << " bank: " << _bank;
    if (payoff > 0) cout << "\tYOU WON " << payoff;
    cout << endl;
    v->Suicide();
  };

  int  _bank, _player;
};

// The SlotCreator is part of the factory that is attached to the top
// and bottom muxes.  It creates one Conduit that contains a SlotState
// for each SlotVisitor it receives and attachtes the Conduit to
// both the top and bottom muxes..
//
class SlotCreator : public Creator {
public:
  SlotCreator(void);
  virtual ~SlotCreator(void);
  // Every Creator has one of these for making new Conduits based on
  // the type of Visitor.  We expect only SlotVisitors.
```

```
  virtual Conduit *Create(Visitor *v)
  {
    Conduit *answer = 0;
    SlotVisitor *s = (SlotVisitor *)v;
    answer = new Conduit("Slot", new SlotState( s->_slot ));
    Register(answer);
    v->Suicide();
    return answer;
  };
};

// The two muxes, top and bottom, each have a SlotAccessor for
// choosing Conduits based on the slot number in a SlotVisitor.
//
class SlotAccessor : public Accessor {
public:
  SlotAccessor(int slots) : _slots(slots) {
    _slot = new Conduit * [_slots];
    for (int i=0; i<_slots; i++) _slot[i]=0;
  };
  virtual ~SlotAccessor(void);

  // Every Accessor has one of these to choose the Conduit that
  // pertains to a Visitor.  We expect only SlotVisitors.
  virtual Conduit * GetNextConduit(Visitor * v) {
    Conduit *answer = 0;
    int slot_num = ((SlotVisitor *)v)->_slot;
    if ((slot_num >= 0) && (slot_num < _slots))
      answer = _slot[slot_num];
    return answer;
  };

protected:
  // Every Accessor has one of these for spreading Visitors to all of/ its Conduits.
  virtual bool Broadcast(Visitor *v) {
    for (int i=0; i<_slots; i++)
      if (_slot[i] != 0) _slot[i]->Accept(v->duplicate());
    v->Suicide();
    return true;
  };

  // Every Accessor has one of these for adding a new Conduit.
  virtual bool Add(Conduit * c, Visitor * v)
  {
    bool answer = false;
    int slot_num = ((SlotVisitor *)v)->_slot;
    if ((slot_num >= 0) && (slot_num < _slots) && _slot[slot_num] == 0) {
      _slot[slot_num] = c;
      answer = true;
    }
    return answer;
  };
```

```
  // An Accessor must defines this to support removal of Conduits.
  virtual bool Del(Conduit * c) { return false; }

  // An Accessor defines this for removing a Conduit based on incoming Visitor.
  virtual bool Del(Visitor * v) { return false; }

private:
  int           _slots;             // number of Conduit * in _slot[]
  Conduit     **_slot;              // array of Conduit *
};


int main(void)
{
  unsigned int __choose_a_better_seed__ = 0;   srand(__choose_a_better_seed__);

  // There is only one simulation kernel.  Tell it that we want to run
  // as fast as possible (SIM_SPEED) for 600 (simulated) seconds.
  Kernel & theWorld = theKernel();
  theWorld.SetSpeed(Kernel::SIM_SPEED);
  theWorld.StopIn(600);

  // Write a record of Visitor travel to the file "casino.vis".
  // You can play back the simulation using the CASiNO Visualizer.
  VisPipe("./casino.vis");

  // Construct the Conduits that make up the slot machine.
  //
  Conduit *coinC      = new Conduit("Coins", new CoinTerminal(6));
  Conduit *topMuxC    = new Conduit("TopMux", new SlotAccessor(3));
  Conduit *factoryC   = new Conduit("SlotFactory", new SlotCreator());
  Conduit *bottomMuxC = new Conduit("BottomMux", new SlotAccessor(3));
  Conduit *tallyC     = new Conduit("Tally", new TallyState(3));
  Conduit *reportC    = new Conduit("Report", new ReportTerminal(10000, 100));

  // Coin to top mux
  Join(A_half(coinC),      A_half(topMuxC));
  // Top mux to Factory
  Join(B_half(topMuxC),    A_half(factoryC));
  // Bottom mux to Factory
  Join(B_half(bottomMuxC), B_half(factoryC));
  // Bottom mux to Tally
  Join(A_half(bottomMuxC), A_half(tallyC));
  // Tally to Report
  Join(B_half(tallyC),     A_half(reportC));

  // Start the simulation.
  theWorld.Run();

  delete coinC;  delete tallyC;  delete reportC;  delete topMuxC;  delete bottomMuxC;
}
```

ACKNOWLEDGEMENT

The work of Abdella Battou and Spencer Marsh was done while at the Center for Computational Sciences of the Naval Research Laboratory, Washington D.C.

REFERENCES

1. Gamma E, Helm R, Johnson R, Vlissides J. *Design Pattern, Elements of Object-Oriented Software*. Addison-Wesley: Reading, MA, 1995.
2. Deutsch LP. Design reuse and frameworks in the smalltalk-80 system. *Software Reusability, Vol. II: Applications and Experience*, Biggerstaff TJ, Perlis AJ (eds.). Addison-Wesley: Reading, MA, 1989.
3. Johnson RE, Foote B. Designing reusable classes. *Journal of Object-Oriented Programming* 1988; **1**:22–35.
4. Mountcastle S, Talmage D, Khan B, Marsh S, Battou A, Lee D. Introducing SEAN: The signaling entity for ATM networks. *Proceedings of IEEE GLOBECOM'00*, San Francisco, CA, November 2000; 532–537.
5. Battou A, Khan B, Marsh S, Mountcastle S, Talmage D. Introducing PRouST: The PNNI routing and simulation toolkit. *Proceedings of IEEE Workshop on High Performance Switching and Routing*, Dallas, Texas, May 2001; 335–341.
6. Brahim GB, Khan B, Battou A, Guizani M, Chaudhry G. TRON: The toolkit for routing in optical networks. *Proceedings of IEEE GLOBECOM'01*, San Antonio, Texas, November 2001; 1445–1449.
7. Turner J, Wu D. Washington University's gigabit network technology distribution program. http://www.arl.wustl.edu/gigabitkits/kits.html.
8. Hüni H, Johnson R, Engel R. A framework for network protocol software. *Annual ACM Conference on Object-Oriented Programming Systems*, 1995.
9. All about Conduits. http://www.glue.ch/~hueni/conduits.
10. Nikander P, Karila A. A java beans component architecture for cryptographic protocols. *Proceedings of the 7th Usenix Security Symposium*, San Antonio, Texas, January 1998.
11. Breslau L, Estrin D, Fall K, Floyd S, Heidemann J, Helmy A, Huang P, McCanne S, Varadhan K, Xu Y, Yu H. Advances in network simulation. *IEEE Computer* 2000; **33**(5):59–67.
12. The Network Simulator—ns-2. http://www.isi.edu/nsnam/ns/.
13. The ns manual. http://www.isi.edu/nsnam/ns/ns-documentation.html.
14. Forouzan BA. *Data Communications and Networking* (2nd edn). McGraw-Hill: Boston, MA, 2001.
15. Hura GS, Singhal M. *Data and Computer Communications: Networking and Internetworking*. CRC Press: New York, 2001.
16. Kurose JF, Ross KW. *Computer Networking: A Top-Down Approach Featuring the Internet*. Addison Wesley: Boston, MA, 2001.
17. Leon-Garcia A, Widjaja I. *Communications Networks*. McGraw-Hill: Boston, MA, 2000.
18. Peterson LL, Davie BS. *Computer Networks: A Systems Approach* (2nd edn). Morgan Kaufmann: San Francisco, CA, 2000.
19. Stallings W. *Data and Computer Communication* (6th edn). Prentice-Hall: Upper Saddle River, NJ, 2000.
20. Comer DE. *Computer Networks and Internets* (2nd edn). Prentice-Hall: Upper Saddle River, NJ, 1999.
21. Keshav S. *An Engineering Approach to Computer Networking: ATM Networks, the Internet, and the Telephone Network*. Addison-Wesley: Reading, MA, 1997.
22. Tanenbaum AS. *Computer Networks* (3rd edn). Prentice-Hall PTR: Upper Saddle River, NJ, 1996.
23. Bertsekas D, Gallager R. *Data Networks* (2nd edn). Prentice-Hall: Englewood Cliffs, NJ, 1992.
24. ATM Forum Technical Committee. ATM user–network interface signalling specification. Version 4.0 af-sig-0061.000, July 1996.
25. ITU-T. B-ISDN ATM adaptation layer—service specific connection oriented protocol. Q.2110, July 1994.
26. Comer DE. *Internetworking with TCP/IP* (3rd edn), vol. 1. Prentice-Hall: Upper Saddle River, NJ, 1995.
27. CASiNO documentation. http://www.nrl.navy.mil/ccs/project/public/casino/.