CASiNO: a Component Architecture for Simulating Network Objects

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

Keywords: telecommunications, protocols, object–orientation, framework, simulators

Abstract

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. Examples of projects using CASiNO can be found at http://www.nrl.navy.mil/ccs/project/public/sean/SEAN-dev.html and http://www.nrl.navy.mil/ccs/project/public/proust.

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. [?, ?, ?]. "Frameworks are becoming increasingly common and important. They are the way that object–oriented systems achieve the most reuse." [?]. In this paper we present a design and C++ implementation of a framework that has been used for modular implementation of several communication protocols and network simulators; notably the User Network Interface for ATM signaling in SEAN (Signaling Entity for ATM Networks) [?] and PRouST (PNNI Routing Simulation Toolkit) [?] developed at the Naval Research Laboratory. 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 define the behavior of the modules within a protocol stack. The Dataflow Architecture is designed to maximize code re-use, 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 delivery and receipt of intra-application interrupts. In the subsequent sections 2 and 3, we shall introduce components of CASiNO. In illustrating their use, we will demonstrate the needs and characteristics of applications developed using CASiNO. Although the focus will be on the abstract base classes which form the foundation of CASiNO, we will from time to time comment on the necessary requirements placed on programmers in using this library. Also for a better perspective, we will sometimes mention abridged details of CASiNO's internal mechanisms and code.

2 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 applications, 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**.[?] CASiNO provides Conduit and Visitor classes; software design using the Data-flow architecture centers around the flow of Visitor objects through Conduit objects. (The term *Dataflow* here is used to mean coarse-grain "packet" flow, rather than classical fine-grain 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

*ITT Systems and Sciences, at the Center for Computational Sciences of the Naval Research Laboratory, Washington D.C.

†Center for Computational Sciences of the Naval Research Laboratory, Washington D.C.

‡University of Southern California, Department of Electrical Engineering - Systems, 3740 McClintock Avenue, Los Angeles, CA 90089-2565. dclee@usc.edu

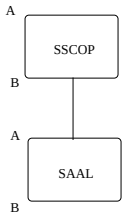


Figure 1: Connected Conduits

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 each other, so too can Conduit objects be connected together. The programmer can connect 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 software architecture of two connected Conduit objects are illustrated by drawing a line between them. Figure 1 illustrates that B side of Conduit object named SSCOP [**?**] is connected to A side of Conduit object named SAAL [**?**].

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 the B side of SSCOP to A side of SAAL, the friend functions are used as the following:

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

The consequence of the Join() is that any Visitor leaving the A side of Conduit SAAL will be passed into the B side of the SSCOP Conduit. Likewise, a Visitor leaving the B side of the SSCOP Conduit will be passed to the A side of the SAAL Conduit. Connecting Conduits in this manner determines possible *communication patterns between the objects during the applications lifetime. Note that the decision of whether to eject a Visitor out of a Conduit's A side, or eject it out of the Conduit's B side, 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 Visitor's passage through a network of Conduit may trigger similar actions. The general manner in which Visitors flow through a Conduit is referred to as the Conduit's *behavior*. Instead of *sub-classing* 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 [**?**]. 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. 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 a*n 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 1-to-1-to-1 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 Conduit's purpose and activity. Finally, (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

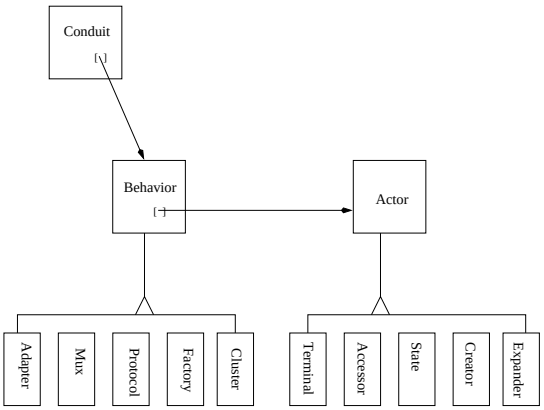


Figure 2: Conduit, Behavior, Actor

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 a 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 because 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" [?, p23].

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, 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. [?] A "whitebox" approach would define the five different types of Behaviors as subclass of the Conduit class, while the "blackbox" approach lets the Conduit have a reference to these Behaviors. A common base class for these five is defined as 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 Actor object as a parameter.

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 five of these methods may optionally be redefined within the applications derived Visitors, though 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 too 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 Conduit's 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.
5. An Expander Actor is required to specify a Cluster.

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

2.1 Protocol Behavior - State Actor

A communication protocol is often specified by a finite state machine. 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 the birth of a Visitor and the sending of that Visitor to a neighboring Conduit. The State Actor is implemented using the state pattern in [**?**].

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);
```

Within the framework, this results in calling the Visitor's method

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

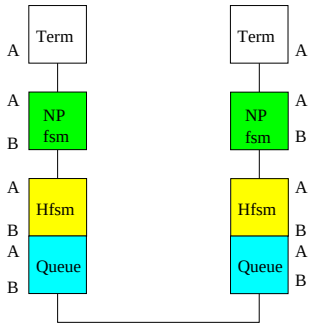


Figure 3: Adapter with a Terminal used to test the PNNI Hello and Node Peer FSMs

passing in the current Conduit's 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 objects Handle method being called:

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

2.2 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.

Figure 3 illustrates the Conduit with an Adapter Behavior and a Terminal Actor, which is named "Term". Note that only A - side of a "Term" is connected to another Conduit.

2.3 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 depart from the switch potentially though 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) field of the cell's header. As another example, the IP protocol layer of a host interfaces both UDP and TCP upper layers. It sends the payload (data) of the received IP datagram to an appropriate upper layer protocol on the basis of "Protocol" [**?**, p92] 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 [**?**]. Accessor class shows in its interface pure virtual function:

```
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, he must include in that derived class a data structure implementing the map. Then the implementation of GetNextConduit(Visitor * v) can look in the map to determine the next Conduit. Figure 6 includes two Conduits with Mux behavior. We represent a Mux Conduit by a trapezoid. In Figure 6, the B sides 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 a Mux and its neighboring Conduit means that the Conduit is known to the Accessor of the Mux. That is, the Conduit is included in the Accessor's map. In Figure 6, Conduits named Call1, Call2, Call3 are listed in the map of the Accessor that belongs to the upper Mux Conduit. They are also listed in the map of the Accessor that belongs to the lower Mux Conduit.

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 4. If a Visitor enters the Mux Conduit from side A, the Accessor's method GetNextConduit(Visitor * v) determines the next Conduit. See Figure 5. If the map does not include a particular Visitor's key, then the Mux sends the Visitor to the default Conduit, the Conduit attached to its side B. 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. See Figure 7. 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.

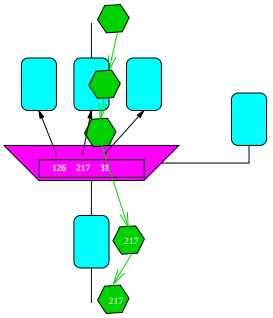


Figure 4: Multiplexing: Visitor gets stamped with the key matching the Protocol it just left

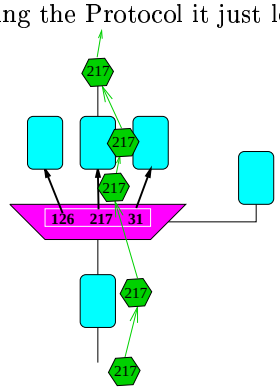


Figure 5: Demultiplexing: Visitor is sent to Protocol with matching key

2.4 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 Fig. 6. A Conduit configured with Factory Behavior is connected to the B side of a Conduit configured with Mux Behavior. Upon receipt of a Visitor, the Factory may make a new Conduit. If it decides to do this,

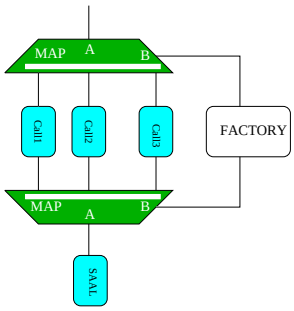


Figure 6: Common usage of two Muxes and a factory

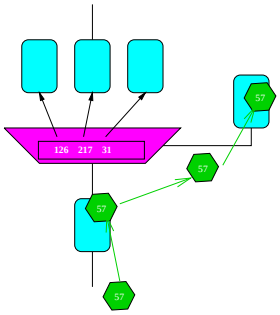


Figure 7: No matching key in the Accessor, Visitor goes to Factory

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 objects on the basis of the identity and contents of the incoming Visitor. After the installation 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 Mux Behavior as in Figure 6, the newly created Conduit is made known to both Muxes.

2.5 Cluster Behavior - Expander Actor

In addition to the four primitive Behaviors of Protocol, Factory, Mux, and Adapter, 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 Cluster behavior specified by this UNI_Call Expander.

In order to specify an Expander, the programmer must declare the network of 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 8 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;
  virtual void OwnerNameChanged(const char *
                                  const name);
protected:
  virtual ~QSAALExpander(void);
  Conduit* _sscf ;
  Conduit* _sscop;
  Conduit* _aal;
};
```

Member variables _sscf, _sscop, _aal are pointers to the three constituent Conduits. As illustrated in this code, the programmer designing a particular Expander must include such member variables in its declaration. The constituent Conduits are instantiated when the Expander is instantiated. The invocation of the Expander's constructor instantiates them. We present 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();
}
```

The two invocations of the Join() function connect the B side of SSCF to the A side of SSCOP and the B side of SSCOP to the A side of AAL, which is illustrated in Figure 8. The constructor of the Expander implements the network structure of the constituent Conduits. In addition to the structure of the constituent Conduits, the programmer must specify A side and B side 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[1]. 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:

[1]A Cluster can only have a one unique conduit as its A interface and one unique conduit as its B interface

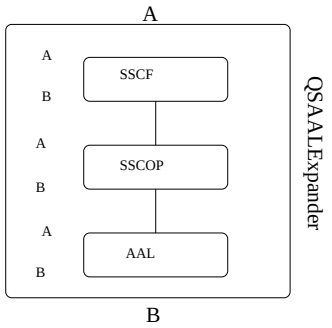


Figure 8: Cluster Behavior

```
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.

3 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 only one Kernel object is 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`.

As mentioned earlier, the "simulation time" 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.

The Kernel object maintains a queue of events, ordered by their expiration times. At each next expiry the Kernel sends a message to the objects, i.e. invokes a user-redefinable operation of the relevant object that needs to be be notified regarding the event. Those objects can then take action in response to the events, 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 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 protected Register() methods to notify the Kernel that it requests the registration of that particular Handler. The Handler class is an abstract base class which 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. Callback() method must be defined by the programmer to fully specify a handler object.

InputHandler and OutputHandler encapsulate a descriptor (file descriptor, socket descriptor, etc.) Input 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/Output Handler in those circumstances. Again, 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. Input Handlers and Output Handlers can respond to any file or device that can be represented by a descriptor (network sockets, files, named pipes, etc.)

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 a SimEntity instance wants to schedule an interruption, it instantiates a SimEvent object, specifying the target SimEntity together with an integer code. It then requests the Kernel to deliver the SimEvent by evoking method

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

This SimEntity method results in the target SimEntity's `Interrupt(SimEvent* v)` method 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. A programmer must write Interrupt method when definig a class derived from SimEntity class.

The 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 method. The destination SimEntity can interrupt in response to the origin SimEntity by using the ReturnToSender method. This method takes a pointer to a SimEvent object and time to deliver, just as the case of the Deliver method. The destination SimEntity can also alter the code of a received SimEvent before sending it back to the origin SimEntity object.

3.1 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:

- 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 been reached yet, then the Kernel checks the devices associated with registered Input and Output Handlers. 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.

4 Visualization of CASiNO Applications

CASiNO includes NV, a tool for visualizing CASiNO simulations. Each CASiNO application can write a log of Conduit and Visitor lifetimes to a file or a pipe. NV displays the simulation graphically on the basis of the logs, showing each kind of Behavior as a different icon and each kind of Visitor as a circle of a different color.

Figure 9 shows a snapshot of the slot machine example of section 5. 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" who will print the result.

5 An Example of a CASiNO Application

The CASiNO application in the appendix 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 10000 coins for the house and 100 coins for the player. There is a payoff whenever the symbols on all three wheels match.

The application, truncated to save space, is shown in the appendix. It 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 simulation can be animated by one of the CASiNO visualization tools.

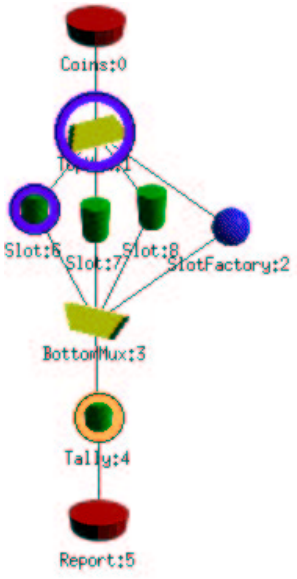


Figure 9: Visualizing the CASiNO slot machine example

6 Conclusion

The modular coarse-grained dataflow architecture of CASiNO, together with its reactor kernel for asynchronous I/O, timers and interrupts, makes it an extremely useful framework for implementating 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:

- Signalling Entity for ATM Networks (SEAN), which implements the host native ATM protocol stack and the ATM User Network Interface, conformant 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 these projects have involved extensive software development by a team 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 ten to a hundred thousand 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 developing network protocols.

References

```
/* 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;
}
```