Heterogeneous Cell-Relay Network Simulation and Performance Analysis with Ptolemy

Allen Y. Lao
Dept. of Electrical Engineering and Computer Science (EECS)
University of California at Berkeley
July 22, 1993

Abstract

Ptolemy is a platform which allows the modeling and simulation of communication networks, signal processing, and various other applications. Its unique set of internal object-oriented interfaces allows it to merge *heterogeneous* descriptions of distinct system components into a unified simulation. This report concerns itself with asynchronous transfer mode (ATM), cell-relay network simulation and shows how the combination of three different *domains* (modes of system description) lends itself very well to this type of experiment: a synchronous dataflow (SDF) domain, a discrete-event (DE) domain, and a message queue (MQ) domain. The work presented follows on the details of a backbone network simulation described in [4]. It is not the goal of this report to focus on network-layer management and related issues; instead, it focuses on modeling techniques and performance evaluation of various popular, practical queueing disciplines for space-division packet switches. Ptolemy's naturalness for the simulation of such a heterogeneous environment will be demonstrated as well as its usefulness for analyzing network behavior and performance.

Memorandum No. UCB/ERL M94/8, Electronics Research Laboratory , College of Engineering, University of California, Berkeley, CA 94720

1.0 Introduction

Future communication networks are expected to support a variety of services handling vastly different data traffic types. The current concept of the *integrated services digital network* (ISDN) allows for such a capability. Three elements are recognized as required for an ISDN: The end-to-end channel is purely digital; the network provides multiple services which differ in terms of required bandwidth allocation; and there is standardization of user access interfaces. ISDN provides a residential customer with two *B* channels at 64 kbps and one *D* channel at 16kbps. [9] These channels can be used for voice and also for data transmissions.

The *intelligent network* model of ISDN primarily seeks to separate the flow of control information from that of user information. Customer premise equipment is connected with *service-switching points* (SSPs) by existing telephone lines known as subscriber loops. The SSP's are connected themselves to *service-control points* (SCPs) by the *common channel-signaling* network (CCS). An X.25 network makes possible the supervision of these SCPs by a management system, operating via packet switching as does the CCS. [Wal] Nevertheless, it is often argued that ISDN's gains over the existing telephone network do not justify its difficulty of implementation since many of its services are already possible with the modern telephone network.

An ambitious alternative to the provision of such multimedia services is the *broadband integrated services digital network* (BISDN). BISDN would make available to residential customers four channels at 150 Mbps, one outgoing and three incoming. These high bit-rate channels would be suitable for providing compressed HDTV and other multimedia information services. Its main design thrust is to economically meet the the widely different requirements of its intended applications with a single network. A key to developing a cost-effective, flexible network is simplification of network architecture coupled with simplification of node processing. The concept of *virtual paths* is integral to the philosophy of *asynchronous transfer mode* (ATM) transmission. Much of the necessity of ATM becomes clear when the inadequacy of *synchronous transfer mode* (STM) transmission in handling the switching of different types of traffic is seen. STM transmission, also known as circuit switching, primarily operates with time-division multiplexed (TDM) frame structures to multiplex transmission channels and paths. But increasing demands for a variety of services with different information transfer rates and a hierarchical scheme of transmission capacities have imposed great complications on TDM-based networking. [5]

The ATM virtual path concept broadly defines a virtual path as a labeled path (bundles of multiplexed circuits) extending between virtual path terminators. The virtual path terminators can identify each connection to or from the network elements included in the segment network to which the virtual path terminators belong. Segment networks may be LAN's or local switching networks. Virtual path terminators could be switching systems or LAN gateways, for example. In addition, the uniqueness of the information unit (cell) which is transported in the network, independent of specific channel rates, allows for straightforward multiplexing/demultiplexing of service channels of varying bit rates. The uniform adoption of the cell as the means of information transport also has the benefit of simplifying hardware and software processing at network nodes. As far as the identification of virtual paths is concerned, this can be done by associating with each cell a *virtual path identifier* (VPI), specifically by attaching it to the cell header. Nodes have their own routing tables which allow them to route incoming cells correctly by means of the cells' VPIs and incoming channel numbers. Routing table renewal at these transit nodes only becomes necessary when paths are set up, released, or rerouted which eliminates processing on a call-by-call basis at each transit node during call establishment and release, thus greatly reducing call setup time. On the other hand, circuit-switched transmissions require a distinct call setup and release phase per call since links dedicated for each call last for the duration of the information exchange and need to be freed up for the use of other callers after the call has ended. The nodes are merely responsible for updating the VPI fields of incoming cells to ensure correct delivery of packets. [5]

The simulation of a BISDN network naturally becomes very complex as one must consider the design of the networking, call processing, and signal processing subsystems. These components, already posing formidable challenges for independent simulation, interact in complex ways. Unfortunately, most simulation tools for design environments impose upon the user a single model of system description. (From this point onward, a single model of system computation/behavior will be referred to as a *domain*.) This limitation suffers from two drawbacks: first, it may not be the most natural domain for the intended user application, and secondly, it prevents the possibility of mixing *heterogeneous* descriptions together into a unified model for simulation which may more suitably describe the system of interest than would any single domain itself. The Ptolemy system, on the other hand, naturally allows for mixed-mode descriptions by assuming little and allowing the user complete flexibility in merging system blocks from arbitrary domains as needed. This is made possible by its internal object-oriented structure.

Ptolemy's usefulness as a simulation vehicle also stems from its hierarchical characterization of modules called *blocks*. A Ptolemaic system is in general composed of a collection of blocks and a *scheduler.* The scheduler determines the order of execution of the system blocks and operates according to the principles of the domain it seeks to emulate. Blocks are, in fact, modules of code which are invoked at run-time, consume data on input terminals, and generate data on their outputs. These input and output terminals of blocks are known as *portholes*. Each connection between a pair of blocks supports a stream of *particles*, which are messages conforming to a general data structure. The hierarchical nature of Ptolemy's system blocks allows for systematic development of useful, general-purpose code modules that can be reused in multiple applications and makes possible the building of more complex systems upon existing ones.

This report describes some areas of focus upon a prototype BISDN model and simulation recently completed, details being given in [4]. It concentrates principally on the realistic modeling of system components as well as practical compilation and analysis of performance statistics. The overall organization of the paper will be as follows. Background information on the three Ptolemy domains that were used to simulate the ATM system of interest will be given to make clear why they were chosen to model it most naturally. Then will follow a brief description of the network-level operation of the simulation, though this was not a primary focus. Next, we discuss a model to realistically characterize and analyze the nature of customer packet voice traffic and models to simulate and analyze efficiency of various queueing mechanisms for space-division packet switches to compare their performances with respect to switch throughput levels and delay. By means of these example applications, Ptolemy's benefits in its support of heterogeneous simulations will be dramatically illustrated.

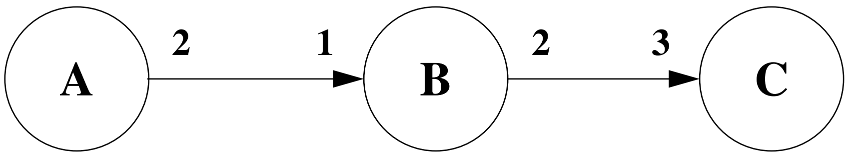
2.0 Overview of Domains Used in ATM Switching Modeling

There are currently many different domains existing in Ptolemy for describing different systems. In the case of the ATM simulation this report addresses, three were utilized: a *synchronous dataflow* domain (SDF), a *discrete-event* domain (DE), and a *message queue* domain (MQ). Each of these three domains will be briefly described with example(s) accompanying each to illustrate simple applications.

2.1 SDF Domain

The SDF domain is especially well-suited for signal processing systems with rationally-related sampling rates throughout. [2] It is a special case of dataflow described by a directed graph of functional blocks, called *actors*, where the arcs in the graph represent streams of dataflow from one actor to another. When an actor “fires,” it consumes some pre-specified number of data samples, called *tokens* (or *particles*), on each input, performs a computation, possibly updating its internal state, and produces a pre-specified number of tokens on each output. The number of tokens produced and consumed must be fixed and known at compile-time, a restriction which explains why the SDF model applies exclusively to synchronous multirate systems. [1] Advantages of SDF are ease of programming (since the availability of data tokens is static and requires no checking), a greater degree of setup-time syntax checking (since sample-rate inconsistencies are easily detected by the system), and run-time efficiency (since the order of block execution is determined at compile-time instead of dynamically at run-time). [2]

The capability to determine at compile-time the necessary order of block invocation for any SDF graph has significant consequences. This “scheduling” information for the SDF graph actors can be passed to a code generator. Thus, it becomes possible to translate signal processing algorithms into efficient assembly code for programmable DSPs. A simple example of an SDF graph and associated schedule is given in Figure 1. The numbers written next to the input and output ports of the actors in the figure indicate how many tokens that terminal either produces or consumes per firing.



Schedule: AAA BBBBBB CCCC

Figure 1 A synchronous dataflow graph and associated schedule

It is interesting to observe above that *one iteration* of the schedule consists of three firings of A, six firings of B, and four firings of C. A user in running the above system would need to recognize what would constitute one “cycle” of the system and then specify the duration of the execution by numbers of iterations. One can easily see that there would be cases when an SDF system’s actors are specified and connected in such a way that determining a system iteration is impossible. In this case, Ptolemy's SDF scheduler would detect the inconsistency in the SDF graph and refuse to execute it.

A typical application of SDF modeling is to filter design and analysis. The block diagram in Figure 2 illustrates an adaptive filter example. It consists of a Gaussian process generator feeding into an FIR filter with fixed, preset taps. The adaptive filter block seeks to track the FIR filter, and the difference between the FIR and adaptive filters is information to be used by the adaptive filter for tap updating. In this example, the Gaussian process generator represents an ouput-only unit, and analogously, the two graph blocks, one plotting both of the filter output signals against each other and the other plotting the difference signal of the two filter outputs, are input-only modules. For the most part, each block is enabled for firing when at least one token is available on each of its input(s) and produces one token on each of its output(s) per firing. For example, the FloatDifference block requires at least one token on each of its two inputs and produces one token representing the difference of the two inputs each time it executes. As for the fixed FIR filter, it has optional decimation and interpolation parameters (both with default one). Thus, its input port would require at least *d* tokens to fire where *d* is the value of the decimation parameter and *i* tokens waiting to be sent on its output to emit data where *i* is the value of the interpolation parameter. By suitable choice of these parameters, it is possible to effectively model systems with multiple, rationally-related sampling rates. It should be noted that the SDF system has *no notion of time*. The two graph blocks, then, plot the values of filter outputs against the sequence number of incoming tokens (starting from 1, 2, ...). Because of its nature, SDF's untimed operation does not lose generality in describing digital signal processing experiments which can be considered to handle data tokens spaced equally in time.

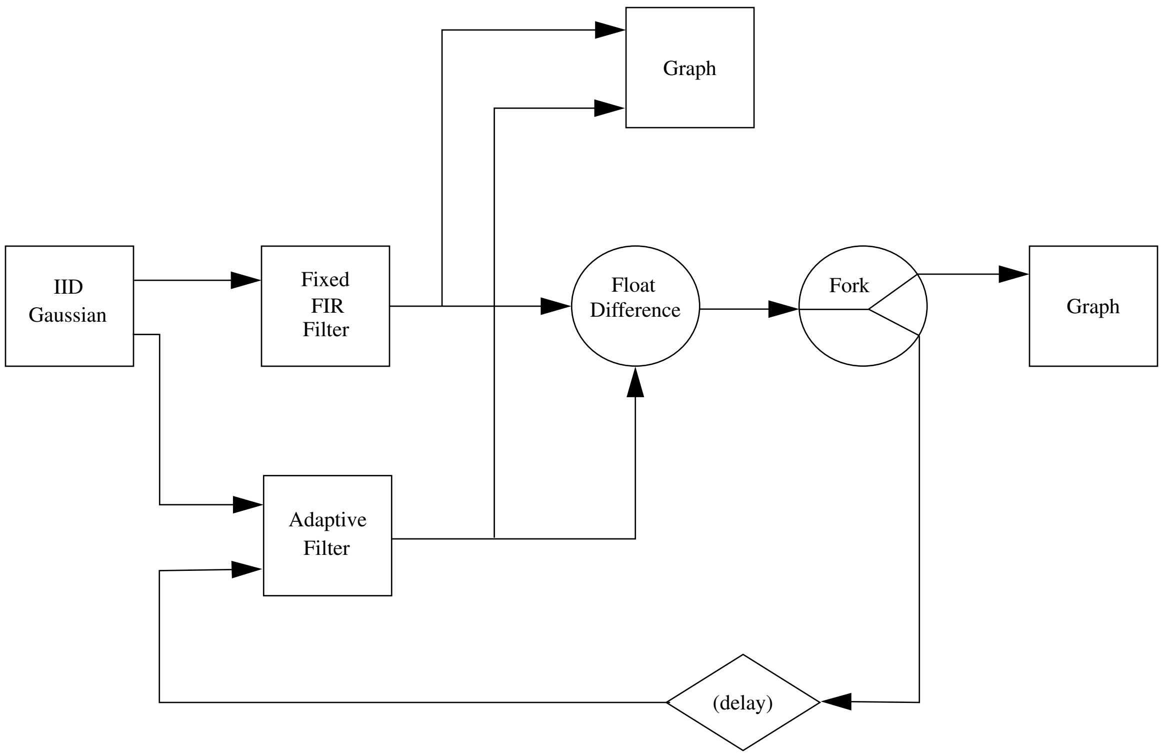


Figure 2 Block diagram of Ptolemy adaptive filter demo

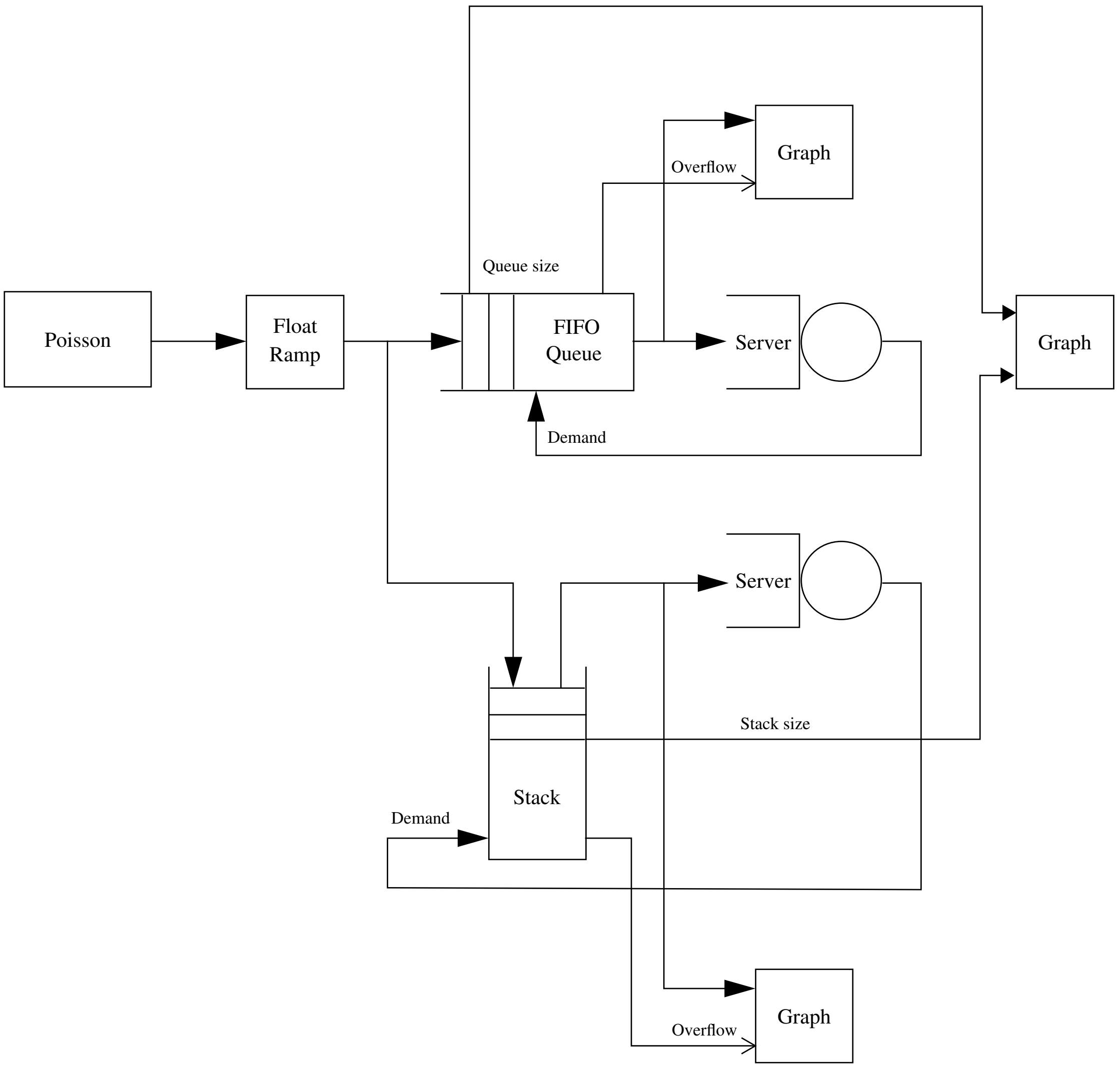


Figure 3 A queue and stack with servers demo in the DE domain

2.2 DE Domain

The DE domain in Ptolemy provides an environment for the simulation of timed systems such as communication networks, queues and servers, and generation of stochastic processes, etc. It operates in an event-driven fashion, processing events by associating with each a time stamp (according to Ptolemy's notion of simulated time; work is being done to realistically model real-time systems as well.) Thus, the DE scheduler manages an event queue to enforce chronological ordering of events. System blocks in a DE graph can generally be classified as either being functional in nature or of delay type. Functional blocks merely perform some operation in “zero” simulated time whereas delay blocks model latency in the system. By appropiate combination of these two types of blocks, discrete-event systems can be accurately described and modeled. Also, naturally, blocks known as “event generators” exist which serve to generate timed processes, such as Poisson processes, independently without requiring inputs for firing. Another example of an event generator would be a system clock.

Figure 3 is a typical system that would be modeled in the Ptolemy DE domain. The Poisson process block is an event generator which produces events at equally spaced points in time, in this case set to unity. These events trigger the FloatRamp block to produce in succession the integers in sequence starting from zero (taking zero time for running upon each invocation since it is a functional block.) These integers are sent to both a FIFOQueue block and a Stack block which operate intuitively and have fixed, pre-selected capacities. Both of these blocks send their outputs to identical server blocks. The servers service incoming events with fixed service times. As can be seen, the server outputs act as demands for the queue and stack blocks. (Initially, the first events to arrive at both the queue and stack pass through without requiring an assertion on the demand input; thereafter, events only exit the queue and stack when the server has produced an output to trigger the demand inputs.) The output events of the queue and stack are plotted together with any overflow events caused by an arrival of an event when queue or stack capacity was full. Finally, the size of both the queue and stack are graphed over time to monitor occupancy. Of course, parameters in this simulation were selected so that the servers’ service times were slightly greater than the mean interarrival time of events to produce overflow conditions.

2.3 Heterogeneity Made Possible - the Wormhole

Ptolemy naturally handles the mixing of different domains in a system by the *wormhole* construct. As has been mentioned, Ptolemy describes systems hierarchically. It recognizes the most fundamental units of computation as elemental blocks called *stars* and from these, allows the building of *galaxies*, which are composed of arbitrary collections of stars and perhaps other galaxies. Of course, it is possible to “flatten” any galaxy into its constituent stars. A self-standing, runnable system of blocks is known as a *universe;* with each universe is associated a particular domain and matching scheduler. Thus, a pictorial representation of, say, a flattened DE universe would appear as in Figure 4. The wormhole ingeniously allows for the enclosure of a certain domain inside a different one, for example, a speech processing application described in SDF nested in an outside, timed DE environment which may model a packet switching network. (Figure 5) To make this possible, the outside domain is given the illusion that the internal SDF universe is merely a DE star! The SDF universe is, in fact, a wormhole in that it encapsulates a completely independent runnable universe with its own scheduler but appears from the point of view of the outside as just a star of that outside domain type. By supporting the use of multiple domains in a single simulation framework, Ptolemy allows a designer great flexibility in modeling system components with those domains which are most natural and straightforward to their implementations.

Actually enforcing the correct operation of such a heterogeneous system is far from trivial and requires correct synchonization of events that communicate across the domain interface. Ptolemy has implemented this by defining a universal event horizon which contains all information necessary for domains to perform their specific operations. In the case of Figure 5, the DE domain has defined a conversion from DE to this universal event horizon and vice-versa. The same is true for SDF and all other Ptolemy domains. Events which pass from the outside DE universe to the internal SDF system are translated first from DE to the universal event horizon, then converted from the universal event horizon to SDF. Correctly processing events passing from the SDF to DE universes should also be straightforward to see. As one can see in following this strategy, having N domains requires definition of $2N$ conversions whereas separately defining a conversion individually from each domain to any other domain would require N squared specifications!

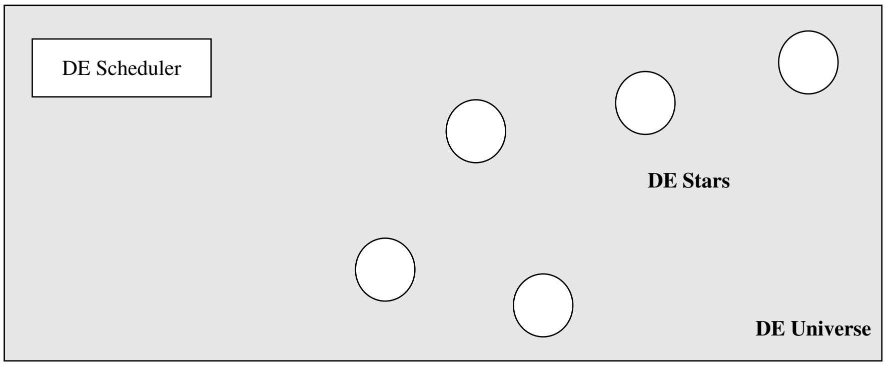


Figure 4 A flattened representation of a homogeneous, DE universe

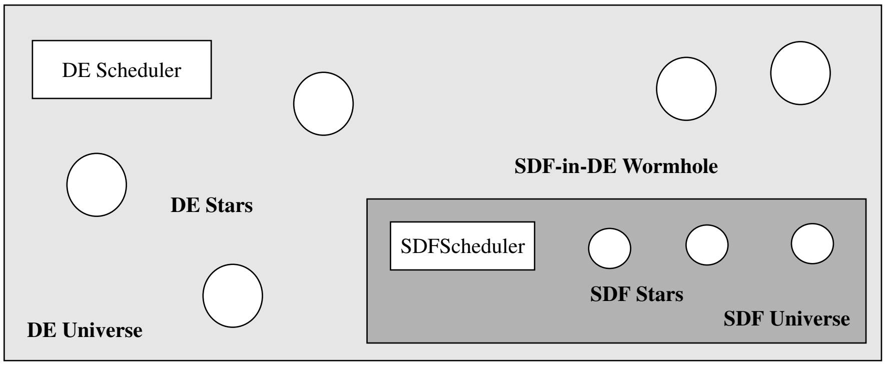


Figure 5 Nesting an SDF universe inside of a DE universe. Ptolemy's capability to do this would effectively model a speech or video processing unit in a switching network. From the point of view of the DE scheduler, the SDF universe appears as a DE star.

There are two key principles of object-oriented programming which allow Ptolemy to achieve its goals: *inheritance* and *polymorphism*. Ptolemy is implemented in C++ and using the C++ language, a user defines various templates or *classes* which consist of associated data members and functions to operate on the data. In a sense, a class is some general data structure enhanced with different functions. These functions are known in C++ as *methods*. Then instances of these class templates can be formed, each of which is referred to as an *object*.

The C++ inheritance mechanism allows for the efficient and flexible formation of new, *derived* classes from already existing ones. A programmer merely needs to add desired facilities to existing class(es), and no reprogramming or re-compilation is required. The root, *base* class from which other classes may be derived serves as a common interface so that objects of these class types may be handled identically in a program. Thus, inheritance provides a natural means of expressing common properties among different classes. [7]

For example, a simple class inheritance diagram in Ptolemy is given in Figure 6. It is very straightforward, showing how domain-specific star classes inherit properties from a base star class.

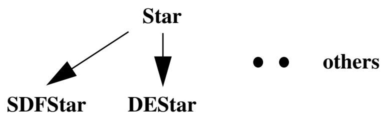


Figure 6 Class Inheritance in Ptolemy

Figure 7 shows how a class can inherit from more than one base class. This is known as *multiple inheritance*. In the diagram, we have derived from class Block the Galaxy and Star classes. The Runnable class is intended to represent at the most basic level any independent, running Ptolemy module. So as an example of multiple inheritance, the Universe class, since it is both a runnable entity and encloses a galaxy, inherits from both the Galaxy and Runnable classes. Note also, how a domain-specific wormhole class (in the figure, XXXWormhole) inherits from both the Star class of the same domain and the Wormhole class to assume the union of their characteristics, the Wormhole class itself being derived from the Runnable class. Naturally, we expect XXXWormhole to have the Wormhole class as a parent, but the explanation below will show why it's also essential for it to inherit from the corresponding Star class, XXXStar.

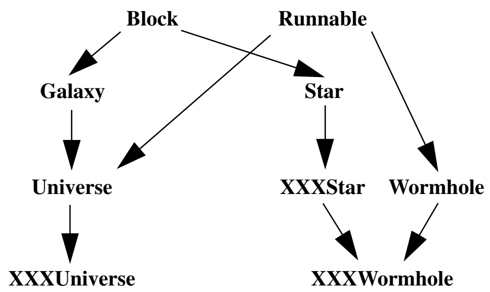


Figure 7 Multiple Inheritance in Organization of Ptolemy Blocks

Polymorphism provides for the *dynamic* binding of class member methods. The method instance invoked is determined by the actual class type of the object. For example, refer to Figure 8. When a scheduler executes a particular "star" (which might either be a star or wormhole), it will call the "go()" method of an instance of type XXXStar where "XXX" refers to the domain in question. In this example, the DE scheduler will not know at compile-time whether this is a genuine DE star or a wormhole. If it is a star, it won't know the specific derived class of the star. Nevertheless, by means of polymorphism, the scheduler will call the correct go() method! Should the star be of type DEImpulse, say, the DEImpulse implementation of the "go()" will be called whereas should it be a DEWormhole, its "go()" would be invoked and proceed to execute the internal domain's scheduler. This transparency between star and wormhole supports heterogeneous simulations.

For a set of classes with a common parent, polymorphism allows their behavior to be customized for any common method as long as its behavior is defined for the parent class, which provides for the possibility that one of the child classes will omit an implementation definition of the method, in which case it assumes that of the parent. New child classes with their own definitions of method(s) they may share with existing classes can be introduced without altering any existing code or the kernel of Ptolemy.

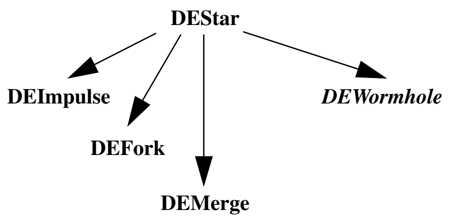


Figure 8 Heterogeneity Made Possible by Polymorphism

2.4 MQ Domain

The MQ domain is targeted at developing control software for applications such as telecommunications call processing software. It views a system as task modules which exchange messages with one another, each message carrying information about actions for the destination task to perform along with necessary parameter information.

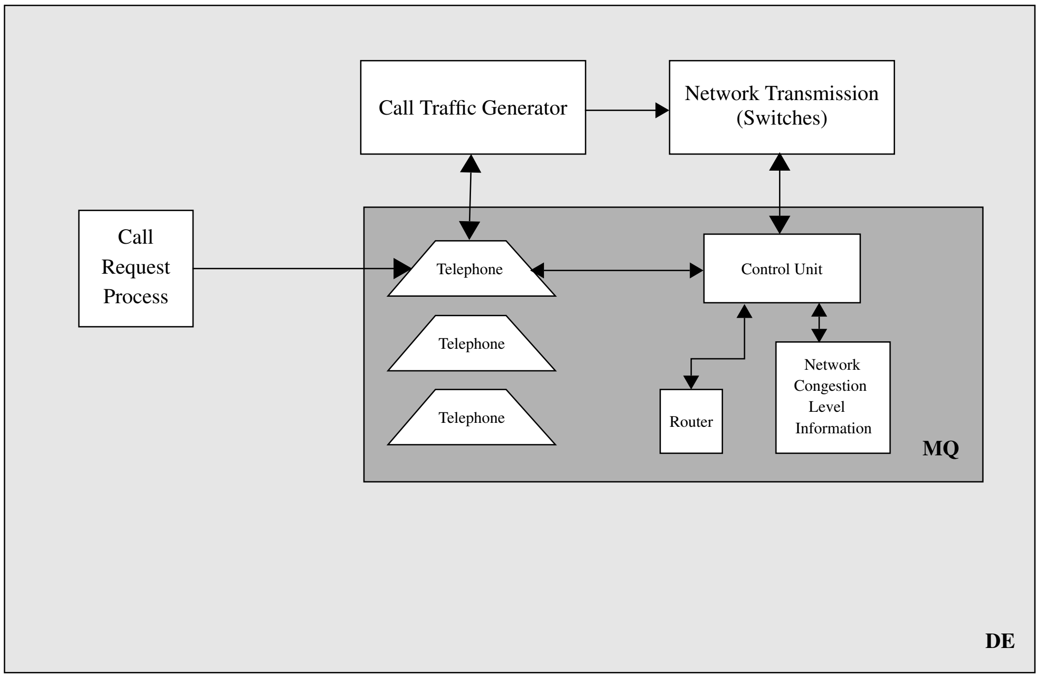


Figure 9 MQ control domain imbedded in DE environment to simulate packet switching network. The diagram shows the connections from the point of view of one customer (one telephone with associated call request process and traffic generator.)

The scheduler has *no notion of system time* and is event-driven, servicing messages at task blocks in a first-in, first-out manner. Blocks in an MQ system always communicate over bidirectional links to enable a client-server relationship over connections. In practice, the tasks act as finite-state machines which take messages as inputs to make state transitions and produce other outputs. This domain also has the capability of dynamically creating and destroying blocks, reconnecting/disconnecting blocks so that it can support dynamic network topologies. In general, the MQ description has little range of use in and of itself but is most natural to simulate software imbedded controllers where an MQ control element may be enclosed in another outside domain external to the MQ system that controls it.

In the case of an ATM network simulation, it would be possible to implement either a form of centralized control by defining a single MQ control element external to the network packet switches or a distributed scheme of control by placing MQ controllers inside each of the switches themselves. Figure 9 shows in a simplified way how a centralized MQ control element would communicate with the rest of the network which resides in a real-time DE environment. The control domain would have tasks to assume the roles of telephones and the control unit itself. An external calling request process of the DE domain would trigger operation of a call request loop via the telephones in the MQ domain. Naturally, the telephone tasks need to be able to trigger generators for telephone call traffic lying outside in DE. The MQ control block itself needs to be able to communicate with the external switches to perform routing table updates, etc. Within the MQ universe, the control unit would have access to a network congestion level information block and possibly a router block as shown to assist it in determining virtual circuits for connecting customers.

3.0 Network-Level Description of Backbone Simulation

The reader may wish to refer to [Loh] for a complete treatment of the backbone ATM simulation with which this report is concerned. This simulation base was primarily conerned with demonstrating the successful simultaneous interaction of the three previously discussed domains - DE, SDF, and MQ - in describing a large-scale ATM system modeling packet switching and call processing control in a certain network configuration. The network topology is shown in Figure 10. It is composed of four switches connected as shown with three customers or CPE's (customer premise equipment) served by each. A centralized controller resides outside the switches and coordinates the actions of the twelve customers which make call requests to random parties. The controller's responsibilities include handling call request loops, requiring the appropiate determination and setup of virtual circuits without overloading switch trunk lines, and ending conversations between callers. All of these actions occur in a simulated, timed DE environment.

There were several networking aspects of this original simulation which were not modeled. Firstly, it did not allow for the possibility of a dynamic network topology with respect to addition or removal of customer nodes or switches in the network. Each switch was modeled as a Batcher-Banyan network (discussed later) in which packets were queued at intermediate nodes of the Banyan network in the event of output port conflict within 2 x 2 switching elements. Naturally, it would be of interest to compare performance in simulating various switching disciplines that differ in their packet queueing strategies. Also of significance is the realistic characterization of source traffic generated by the customers during a phone call. Much of the rationale behind the ATM virtual circuit switching discipline is the effective handling of "bursty" arrival processes. The original switching network model did not simulate realistic traffic processes but concentrated on the correctness of call processing in establishing virtual circuits to ensure proper delivery of packets.

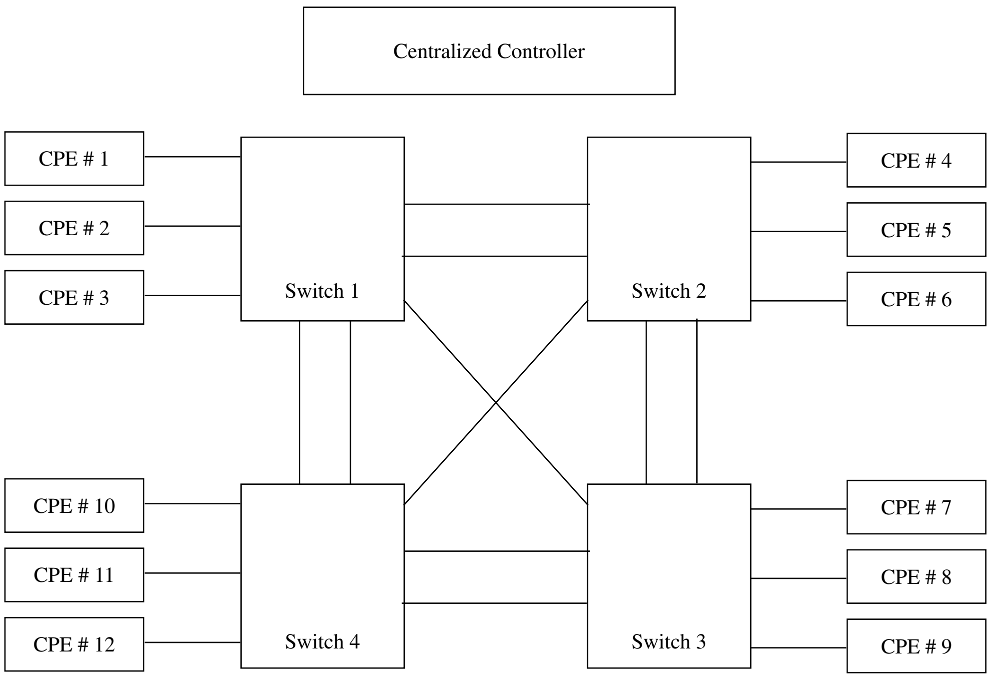


Figure 10 Network topology of backbone simulation

What follows is a more low-level look at the implementation of the centralized network controller and the operation of the packet switches in the original backbone simulation.

3.1 Control Operation

Referring back to Figure 9 shows the basic anatomy of the control domain's structure and interaction with the external, timed DE domain. Modeled in the untimed, task-oriented MQ environment, the control domain consists of a number of telephones (in this case twelve to represent each of the customers in the system), the control unit, and two stars to assist the control unit in its functions, a router and a network congestion level table. Several other stars in the DE domain which communicate directly with stars in the MQ control element are also pictured. Each telephone is unidirectionally triggered by a separate, independent call request star and also communicates bidirectionally with a distinct traffic generator star. In addition, the control unit communicates bidirectionally with each of the switches in the system, in the case of this network topology, four. Within the MQ domain, in which all connections are two-way, the control unit communicates with each of the telephones and also connects with the router and congestion level table.

Having pointed out the structural details of the network control, the mechanics of its operation are now explained. In actually implementing the MQ-in-DE wormhole, Ptolemy runs the DE system in its usual chronological fashion and will at some point, process an event which passes from an external DE star to an internal MQ star. Recall from section 2.3 that Ptolemy achieves this by first providing for a conversion from DE to a universal event horizon and then from the universal event horizon to the MQ description. MQ has no need for timing information and so disregards time stamps of any token(s) entering the system from the outside. In deciding whether to run an internal wormhole scheduler, Ptolemy checks if the internal domain is "ready" to run; in this case, MQ is always ready to run and is always successfully triggered. At this time, the MQ scheduler begins operation in processing the triggering event(s) appropiately. MQ stars pass messages to one another until the scheduler sees that there is a deadlock condition, meaning that there are no more messages to be processed in the system. Control is now returned to the external DE scheduler. During the execution of the MQ control, messages or tokens may be produced which are passed from MQ star(s) to external DE star(s). Since the MQ domain has no notion of time, the DE environment views the execution of the MQ system as occurring in zero elapsed time, to some extent reflecting the usual negligible time duration of control functions. Thus, time stamps(s) of tokens produced by MQ which are output across the domain boundary are identical with those of token(s) that originally were passed from DE into MQ in triggering the operation of the MQ universe.

During the course of the simulation, the call request process stars of the DE domain generate events according to independent Poisson processes to model requests which are fed to the MQ telephone stars. These requests contain information about the desired customer party to be called, and subsequently, the telephone star proceeds to interact with the control unit as the control attempts to set up the call. Should the called party be already engaged in another conversation, the telephone is notified of failure in the call attempt; otherwise, the control unit determines trees of possible virtual circuits by which the two customers can communicate with the aid of the router star. The reason for this is that switch trunk lines have a maximum allowable utilization which is a parameter specifiable by the user. By successive inquiries of the congestion level table, the control unit can establish a suitable virtual circuit to support the conversation. It will attempt to set up "direct" conversations over more roundabout ones; for example, referring to Figure 10, a direct conversation between CPE #2 and CPE #8 would only employ switches 1 and 3. If a packet sent from CPE #2 to CPE #8 was following a more indirect route, it would first traverse switch 1, then either switch 2 or switch 4, and lastly switch 3. The control unit allows as a worst case the traversal of two "intermediate" switch nodes for a packet to travel from source to destination, the previous scenario considering one "intermediate" switch. Should no suitable route be found, the calling telephone is notified of failure. Otherwise, the control records the relevant information pertaining to the newly established conversation, updates the appropiate switches to implement the virtual circuit, modifies the entries of the congestion level table, and informs the called telephone of the new connection.

Finally, the two telephones are told to begin conversation, whereupon, they signal their corresponding call traffic generator stars in the DE domain to begin transmitting data. These actions constitute the call request loop of the control domain.

Both traffic generator stars have a specified length of time to transmit data as requested by the telephone stars. The one which finishes first informs its telephone that it has completed transmission, initializing a call termination loop in the MQ domain. The telephone sends a call termination request to the control unit which proceeds to order the other telephone in the conversation to request its associated traffic generator star to cease transmission. Once this is accomplished, the control updates the congestion level table accordingly and records the terminated conversation.

In considering the design philosophy of the MQ domain, one can see in this simulation context the need for its stars to effectively communicate messages with each other and with stars lying in an outside "driver" domain which view the MQ universe as an embedded controller. Thus the domain supports bidirectional connections among its elements as well as one-way or two-way connections with outside domain stars. Figure 9 shows that a call request star communicates unidirectionally with a telephone whereas a telephone's interaction with a traffic generator star needs to be two-way. Also implicit is the naturalness of connecting multiple instances of a certain star type to another single star; in this case, the control unit needs to communicate with twelve telephone instances. In order to support new or failed customer nodes, the capability to alter the block toplogy within the domain is crucial, which MQ supports. MQ also lends itself naturally to efficient system state representation. Whenever execution is surrendered to the outside domain in the middle of a processing sequence, system state determines the actions taken once the MQ is reinvoked. In any large-scale simulation such as this one, there will be sophisticated interaction between a controller and the rest of the embodying network so this needs to be effectively and efficiently modeled.

3.2 Switch Operation

Space-division packet switching is playing a crucial role in the development of high-performance integrated communication networks for data, voice, image, and video and multiprocessor interconnects for building highly parallel computer systems. Unlike current packet switch architectures with throughput maximums measured in the 10's of Mbits/s, a space-division packet switch can achieve throughput levels in 1's, 10's, or even 100's of Gbits/s. These capacities are attained by the use of a highly parallel switch fabric coupled with simple per packet processing distributed among many high-speed VLSI circuits. [3]

A space-division packet switch can be viewed as a box with N inputs and N outputs with the function of routing packets arriving on its input terminals to the appropiate outputs. The routing information necessary to transport packets correctly is often contained in the headers of the packets themselves; in other words, the switch is self-routing by nature. Buffering strategies for the switch are an important consideration as packets may need to wait for certain desired connections to be made available. The buffer location and capacity depend on the switch architecture and nature of incoming traffic. [3]

There are different types of space-division switches; of interest here are those based on multistage interconnection networks composed of 2 x 2 elementary crosspoint switches, each of which can assume one of two states, a cross state and a bar state. Consider Banyan-based switches, an example given in Figure 11 known as an OMEGA network. A simple procedure allows for correct routing of packets from input to output by considering the binary representation of the output port address $(b_1 b_2 \ldots b_k)$. Given that the packet has reached stage s of a switch, it should be routed to the upper output if $b_s = 0$ and to the lower output if $b_s = 1$, regardless of which input to the elementary switch the packet may have arrived at. This demonstrates the self-routing aspect of the Banyan switch structure. It turns out that the connection of N inputs to N outputs requires $(N/2)\log_2 N$ elementary binary switches [8], as in the case of Figure 11 where $N = 8$ and a total of twelve switching elements are necessary.

A switch is denoted as *blocking* if it cannot always be assured that there will always be enough switch points to provide simultaneous, independent paths between arbitrary pairs of inputs and outputs. The Banyan switch shown in Figure 11 is an example of a blocking network. In a Banyan switch, even when every input is assigned to a different output, as many as $\sqrt{N}$ connections may contend for use of the same center link. Logically enough, the use of a blocking network should only be considered as a feasible option under light offered load conditions or if the switch can be run at a much faster speed than the input and output trunks. [Karol] To resolve this performance limitation of a mere Banyan switching arthitecture, queueing at the inputs to the switch can be implemented, along with a Batcher sorter, shown in Figure 12, that sorts input packets by destination address in order to remove output conflicts and offer the Banyan network (sub)permutations that are guaranteed to pass without conflicts. Packets not initially selected are buffered for later attempts. The tandem combination of the Batcher sorter with the Banyan network is hence *nonblocking* and known as the Batcher-Banyan switching fabric, the switching structure that was modeled in the prototype simulation.

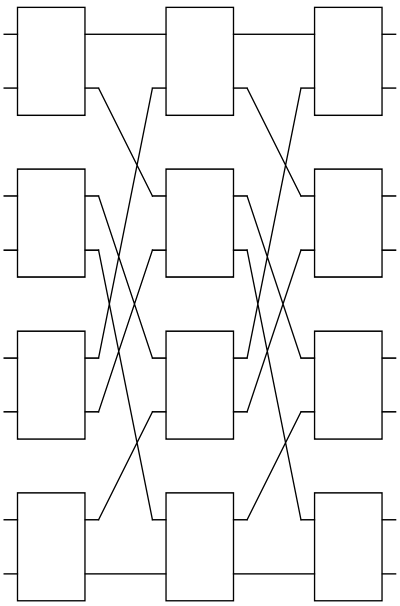


Figure 11 8 x 8 Banyan Network.

The four switches in Figure 10 were built in the SDF domain primarily because of the suitability of SDF for describing synchronous operations. In the case of the Batcher-Banyan switches, incoming trunks can be considered to divide time into equally spaced slots. For each trunk, then, at the beginning of each time slot, the first packet on the input line is accepted into the switch should packet(s) be available. Recall that SDF yields efficient scheduling because of its ability to determine system block execution order at compile-time. Each switching element consists of two input ports and two output ports. When one element fires, it consumes one token on each of its two inputs and outputs one token on each of its outputs. Of course, in practice, because of the irregular nature of incoming traffic, packets will not necessarily occupy both inputs of a crosspoint upon firing or both of its outputs after firing. In such a case, a “null” packet is placed to satisfy the SDF domain requirements which might seem somewhat unnatural. Nevertheless, the efficiency gains of SDF added to its natural characterization of the synchronous aspect of the switching operation are quite notable. Most switching mechanisms when described in SDF lend quite well to possible hardware implementations.

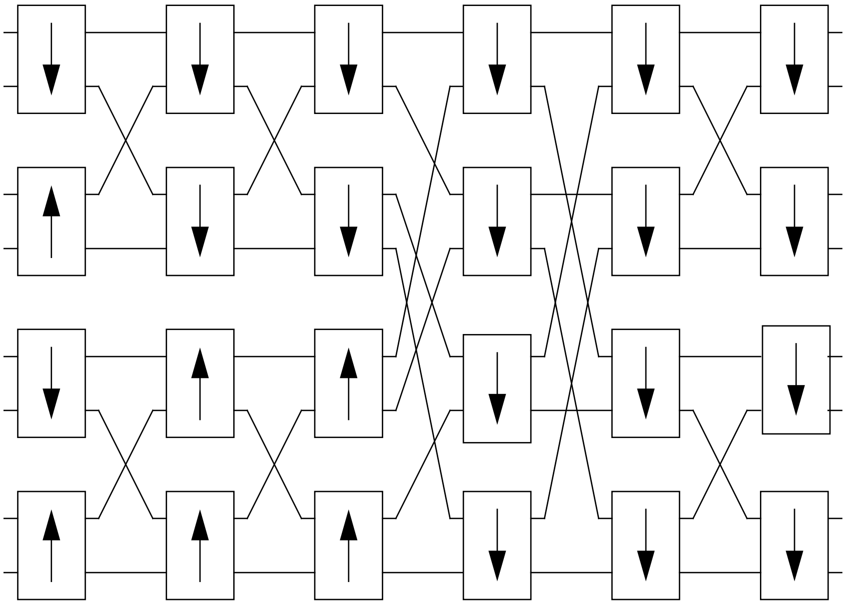


Figure 12 8 x 8 Batcher Network.

The behavior of an SDF-in-DE wormhole is quite intuitive. Since SDF is an umtimed domain, once again as in the case of MQ, each execution of the enclosed SDF system from the external DE's perspective takes zero time. In order for the internal SDF system to execute, it must be provided with at least one token on each of its input ports that lie on the domain interface with DE. For this reason, before each of the SDF switches, a DE synchronizer star was provided which examined input trunk lines for the switch upon request of a periodic demand signal simulating the start of new time slots. The synchronizer either passed a single available packet through or if none were available, provided a null packet. In this way, the SDF-in-DE wormhole would always have at least one token on each of its inputs and always be able to execute to process packets arriving on the switch inputs. Conversely, a star was placed at the outputs of the SDF switch for the process of "filtering" out any null packets that might happen to be produced on the switch outputs, amounting to output trunk(s) not being utilized for that clock cycle. These two simple stars enabled the DE and SDF domains to interact in realistically characterizing a packet switch, both of them exhibiting strength and naturalness of description with regard to what blocks they represented.

4.0 Programming Network Control with Message Queue Domain

Having discussed the nature of our simulated ATM network's control and switching elements, we now examine in greater detail the problem of programming the stars comprising its centralized control unit. As already mentioned, this work was done in the MQ domain. Prior to the development of our simulation, the SDF and DE domains had already been well-established in Ptolemy, but the MQ domain was newly developed for the express purpose of facilitating the programming of network control. What follows is, first, a look at the MQ domain itself and secondly, specific coding examples of MQ stars used in our simulation.

4.1 Design of the MQ Domain

General comments on the semantics of the MQ domain were given in section 2.4. As mentioned, an MQ universe consists of stars that pass messages with one another over bidirectional connections. In designing the domain, one would want to ensure correct operation of the system regardless of the order in which stars are fired. Besides this guiding principle, we would not want to stipulate for a given star currently being executed in what order we step through its portholes. The only assumption that is to be made about the order of processing messages is that messages arriving at a star's input porthole over a given link are to be serviced in the order they arrive. By adhering to these principles, we can straightforwardly implement the MQ scheduler.

The MQ domain provides various methods for MQ-type porthole (that is, portholes belonging to a star in the MQ domain) classes consistent with the above conditions that allow for the reasonably efficient coding of MQ stars. An MQ porthole supports two-way connections unlike other domains' portholes in Ptolemy which generally support unidirectional connections. For that reason, it can be considered as the union of two "ordinary" portholes, one for incoming messages and one for outgoing messages. Following are the main methods for the *MQPortHole* class along with brief explanations to their functions. Necessary parameters are listed in parenthesees following the methods' names.

- *expect ()* — indicates that the star expects its next arriving message to arrive on this particular port; scheduler will flag an error if next message does not arrive on this port
- *block ()* — says the scheduler is to prevent the star from detecting new, arriving messages on this port, messages arriving at a blocked port remain queued at the port, restore to normal with *unblock ()* method
- *flag ()* — raises a flag for this port, star can check on status of a port's flag by calling *flagged ()* which returns a Boolean result, call *unflag ()* to restore original state
- *in (int)* — returns a reference to a particle for the incoming porthole with the argument representing the delay of the access, i.e. an argument of zero gives the most recent token
- *out (int)* — same as above but for outgoing porthole
- *newMessage ()* — indicates whether a new message has arrived on the incoming porthole, returns a Boolean result

In the MQ domain, there also exists a class called *MQMultiPort*. In certain situations, an MQ star will have a set of portholes that serve identical and parallel purposes, in many cases connected to multiple instances of another MQ star. It is convenient from a programmer's point of view to be able to refer to this set of portholes as an aggregate whole, which motivates the existence of "multiportholes," represented by the *MQMultiport* class. For example, referring to Figure 9 from earlier, we see that the star representing the control unit is connected to multiple instances of a "telephone" star, one per customer node. Thus, a multiporthole would be declared for the control star to connect to the parallel instances of the telephone star. Following are methods provided for the *MQMultiport* class:

- *expect ()* — calls *expect ()* for each of the member portholes, *unexpect ()* similarly
- *block ()* — calls *block ()* for each of the member portholes, *unblock ()* similarly
- *flag ()* — calls *flag ()* for each of the member portholes, *unflag ()* similarly
- *port (int)* — returns reference to the individual porthole whose index within the multiporthole is given by argument

The *MQStar* class itself has a method *portToService()* which returns an MQ porthole that has an unserviced message, if any. It also has a member called *state* which the user can use to record the current state of a star, as an integer.

4.2 Examples of MQ Domain Star Code

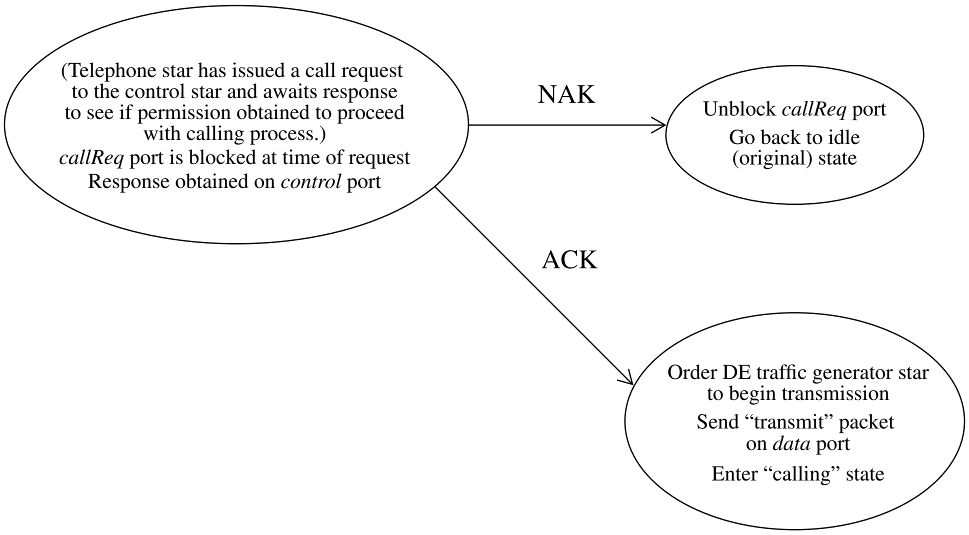


Figure 13 Actions taken by MQ telephone star after issuing call request to control star

Typically, stars in the MQ domain are coded as finite-state machines. The MQ scheduler calls on stars in succession, firing them by executing their *go()* methods. A star's *go()* method will fundamentally check upon its current state, branch to its prescribed set of actions for that state, update its notion of state, and possibly output particles. Stars were developed to implement the centralized network control of the backbone ATM network that utilized the methods described in the previous section. A few examples of code segments will most clearly indicate the nature of the star programming.

Let's consider the *MQTelephone* star. Refer back to Figure 9. It has portholes named *callReq*, *data*, and *control*. The *callReq* port is connected to the call request triggering star in the external DE universe. The *data* port communicates with the external DE call traffic generator star. Lastly, the *control* port connects the star to the control star in the telephone's own MQ universe.

Figure 13 provides a flow diagram indicating what actions are taken by the telephone star after issuing a call request to the control star. It has been triggered into execution by a particle arriving from its external call request star on the *callReq* port. While it remains in its call request loop, it cannot further process additional call requests so its *callReq* port has been blocked. Awaiting approval from the control star, it has the possibility of receiving either a positive acknowledgment (ACK in Figure 13) or a negative acknowledgment (NAK). In the former case, it can order its associated DE traffic generator star to begin transmission. Otherwise, it can safely unblock its *callReq* port and head back to its original idle state.

From the flow diagram, it becomes a simple matter to translate this into pseudocode that draws upon the MQ porthole methods of the previous section. The telephone star's *go()* method appears as this:

```
switch (state of Telephone star) {

    case (idle state): . . .

    . . .

    case (issued call request):

        Access incoming message by "control.in(0)"

        if (message is ACK) {
            send transmit message to DE traffic generator,
                access output token by "data.out(0)"
            data. expect()
            control. expect()
            state = calling state
        }

        else if (message is NAK) {
            callReq.unblock()
            callReq.expect()
            control.expect()
            state = idle state
        }

        else {
            error message
        }
```

```
                end case

            . . .

            case (calling state): . . .

            . . .

        }   /* end switch */
```

The actions taken in the code mirror those of the flow diagram. In the "ACK" branch of the *if-elseif-else* structure, *data. expect()* and *control. expect()* are called because the star enters the calling state and anticipates one of two possibilities. Possibly, its traffic generator will later order it to cease transmission in which case it receives a corresponding message on its *data* port. Also, it is possible the other phone will end the conversation first in which case that phone would send a message to the control star, and the control star would send a "stop" message to this phone over the *control* port. In the "NAK" branch, *callReq. expect()* and *control. expect()* are called since once again, this telephone star will be fired in one of two possible situations. First, it may receive a message from its call request triggering star over the *callReq* port. Or it may be randomly selected by the control star to converse with another customer that has issued its own request for a conversation.

Let's consider a more complicated example: the *MQControl* star in its "idle" state. Refer to Figure 9 to see that this star has connections with every other star in the MQ universe in addition to the network's switches. (It is actually connected to each of the multiple Telephone star instances though the diagram doesn't show this.) For the purposes of this example, we only are concerned with the Control star's connections to the Telephone stars via its multiporthole *phones* and the network control congestion level table via its porthole *mqcclt*. The pseudocode for the Control star's actions in the idle state:

```
switch (state of Control star) {

    case (idle state):

        Access incoming message with "portToService()"

        if (message is new call request) {
            record identities of calling customer and randomly selected "target"
                customer as vciActive and vciPassive (from message)
            issue inquiry message to congestion level table star to determine
                whether target customer is "busy," access output token
                by mqcclt. out(0), anticipate either ACK or NAK
            phones. flag()
            mqcclt. expect()
            state = beginning of call request loop
        }

        else if (message is call termination request) {
            from message, determine identity of  customer desiring to end
                conversation and access internal data to determine other
                customer's identity.  Record these respectively as vciActive
                and vciPassive
```

```
                    issue "stop" message to other customer, access output token by
                            phones. port(vciPassive). out(0)
                    phones. flag()
                    phones. port(vciPassive). unflag()
                    phones. port(vciPassive). expect()
                    state = beginning of call termination loop
            }
            else {
                    error message
            }
            end case

        case (beginning of call request loop): . . .

        case (beginning of call termination loop): . . .

        . . .

}   /* end switch */
```

When the Control star is sitting idle, it may be triggered into action by one of two means. First, it may receive from a Telephone star a call request message containing the identity of the customer issuing the request and a randomly selected customer it has chosen to establish a connection with. This information is locally recorded as *vciActive* and *vciPassive*. The Control star is responsible for checking the network congestion level table to see if the *vciPassive* party is already busy in conversation. Thus, it produces a message on its *mqcclt* porthole containing the *vciPassive* value and calls *mqcclt. expect()*, anticipating an ACK or NAK message on this port as appropiate the next time it is invoked. While it is engaged in this call processing loop, it cannot also process additional requests from other telephones; hence, it flags all of its connections to telephone stars by making the multiporthole method call *phones. flag()*. If it later finds from the congestion level table star that the *vciPassive* party is free to talk, it will "unflag" the two ports connecting to the two relevant telephone stars. After the Control star has unflagged these ports, there is no problem when one of these phones wants to end their session as it can safely issue a call termination request message as described below. At this point, the Control star proceeds to the call request loop.

The second method by which the Control star might be invoked from the idle state is receiving a call termination message from a Telephone star. The message will contain the identity of the customer wishing to end a conversation which is noted as *vciActive*. Thus, the Control star can check its internal data to see which party it is conversing with (marked as *vciPassive*) and then to send that party a "stop" message. That customer will go about asking its associated traffic generator star to stop transmission and then to send back to the control an ACK message. After the control has sent the message, it flags all of its connections to telephone stars for the same reasons as in the preceding situation except, of course, the customer whom has been asked to cease transmission since an ACK is expected from it. Thus, the star makes method calls *phones. flag()*, *phones. port(vciPassive). unflag()*, and *phones. port(vciPassive). expect ()* in succession before proceeding to the state corresponding to the beginning of the call termination loop.

It is also of interest to consider what actions the Control star should execute when a message is received from a telephone over a port which has been flagged. A call request message from a telephone is bluntly replied to with a NAK message from the control. However, the same cannot be done for a call termination message; otherwise, the telephone that issued the message will be left dangling indefinitely. Thus, the message is queued up for later service.

When the control has completed the call request or termination loop which it is currently engaged in, it will check this queue for possible call termination requests that it can service. This process repeats until the queue is empty.

5.0 Characterization of Source Traffic

An issue of interest in the field of network traffic study is the behavior of multiplexers for multiple, independent arrival processes. Interestingly, the aggregate packet arrival process resulting from the superposition of the streams from multiple voice sources is complicated, possessing a burstiness (high variability) that can lead to unexpectedly large packet delays in a muliplexer under heavy load conditions. [6] Thus, a reasonable issue to investigate is the modeling and characterization of such an aggregate arrival process.

5.1 Single Voice Source

The inherent complexity in a multiplexed voice stream stems from the individual burstiness of a single voice source. A single voice source alternates between periods of talking activity and silence. Periods of silence produce no packets at all whereas a talkspurt produces packets at fixed intervals according to a packetization period. Another way of looking at this is that the interarrival times are usually equal to one packetization period but occasionally longer (one packetization period plus a silence period.) [6] Thus the output of a single voice source is a renewal process, i.e., one with i.i.d. (independent and identically distributed) packet interarrival times. But as mentioned, the interarrival-time distribution of this one source is highly variable, so that as compared with a normal Poisson process, the voice process is quite bursty.

As modeled in Ptolemy, a voice source specifically alternates between silence and talking periods. A silence period is exponentially distributed with some mean . A talkspurt produces packets at fixed intervals of T ms and generates a number of packets geometrically distributed over the positive integers, a very significant assumption. In the case of this simulation context, T was chosen to be 16 ms and a silence interval was of mean 650 ms. The mean number of packets produced in a talkspurt is 22, or 352 ms. So for the source, a packet interarrival time assumes value T = 16 ms with probability $p = 21/22$ and value 666 ms with probability $1 - p = 1/22$.

A simple system to graph the voice process is shown in Figure 13, constructed in the DE domain. The FloatRamp star merely counts upward from zero when fed an input event from the Voice star, and this information is graphed in the Xgraph star. Secondly, the actual output of the system is displayed in Figure 14. In it can be clearly observed the periods of call activity made of packets equally spaced in time with separating gaps of silence.

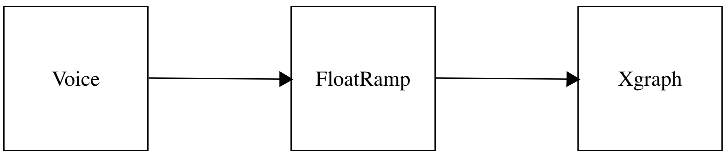


Figure 13 System to display a voice process

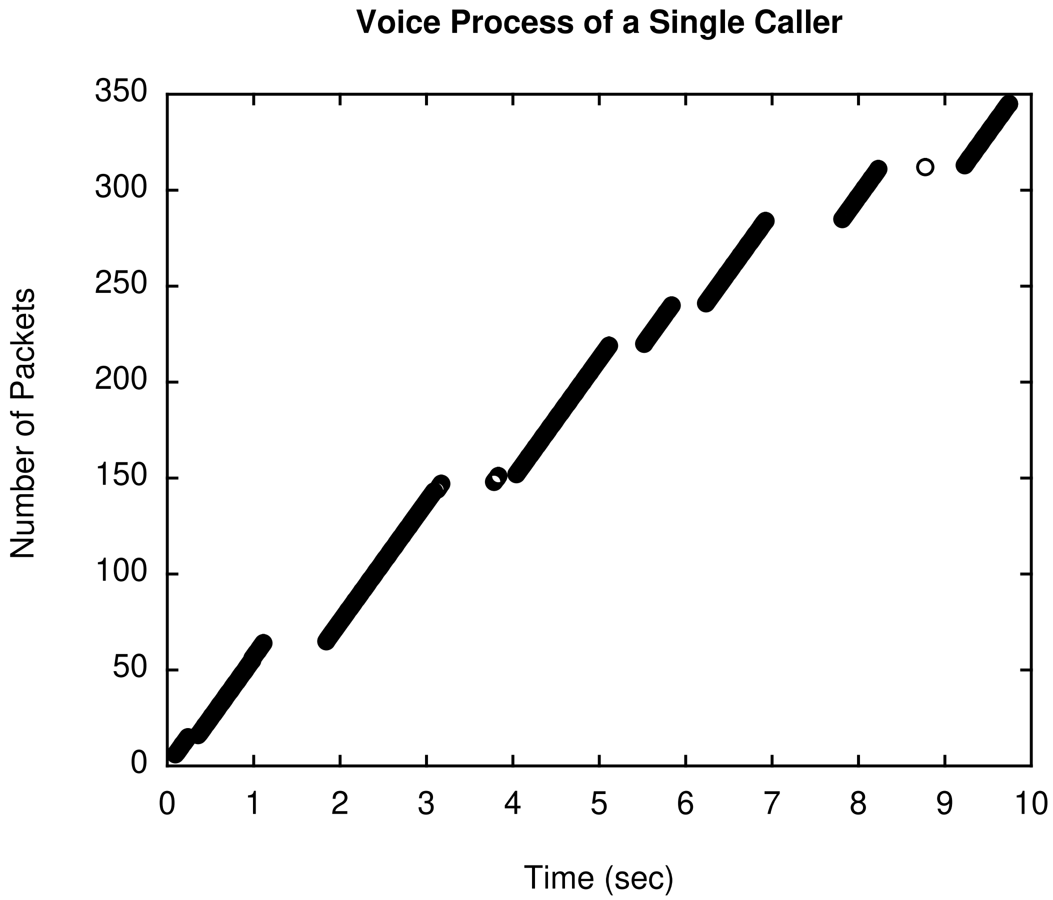


Figure 14 Output of system in Figure 13

Figure 15 displays the results of the simulation described in section 3.0 structurally illustrated in Figure 10, in the case where each customer, when active, transmits according to the single voice source model just described. The graph shows the calling activity of CPE nodes 5 and 6, plotting their transmitted and received packets with an approximate scatter plot. Markers in the plot are spaced for legibility with respect to the time axis since the packetization interval of 16 ms is so small in relation to marker size. The vertical axis tells which one of the twelve CPE nodes is being communicated with. Of course, graphs for the other CPE's can be obtained to cross-check the transmitted and received packets for two CPE's communicating over some interval. In the figure, for example, we can see that CPE #5 and CPE #6 talk to each other for approximately three seconds at around time 30 (seconds.)

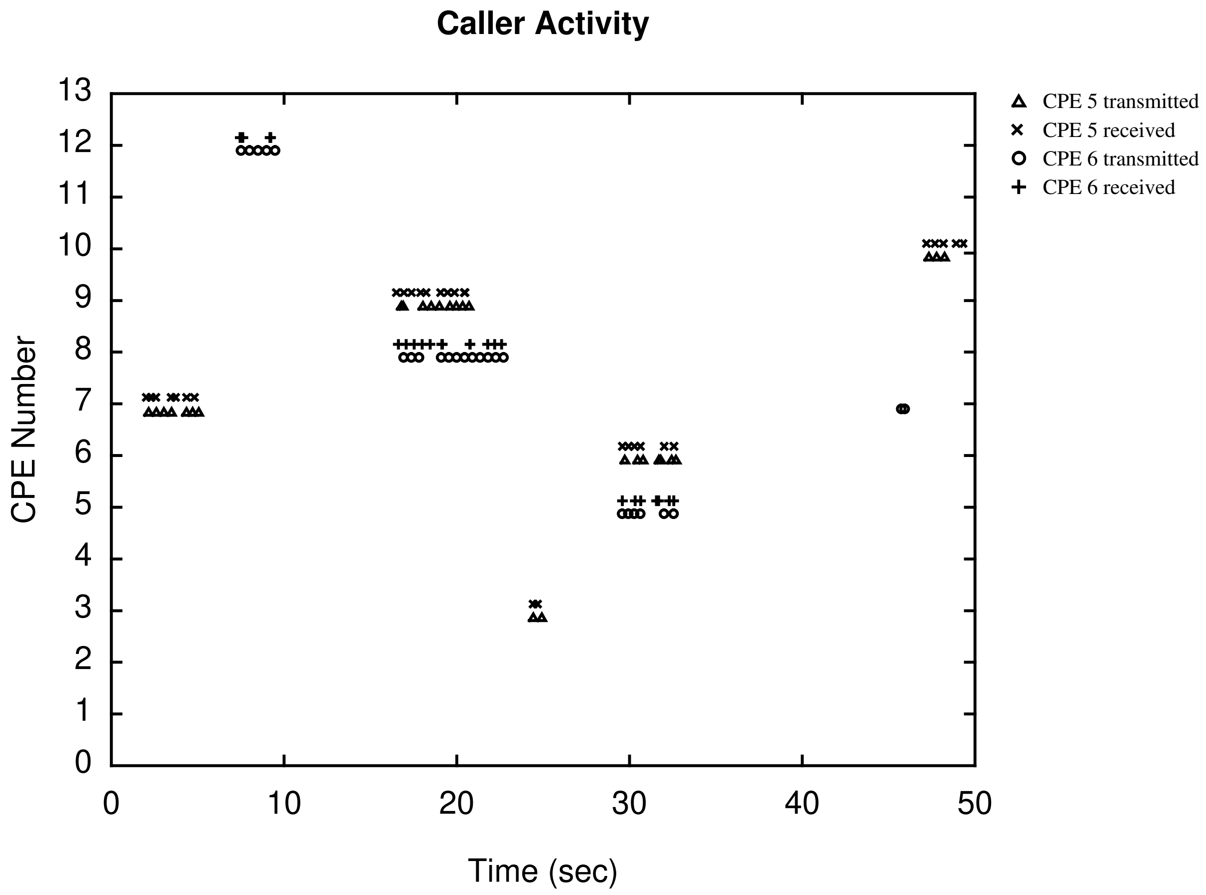


Figure 15 An approximate scatter plot monitoring calling activity for two CPE's, each customer transmitting according to a single voice source model.

5.2 Multiple Voice Sources

To study the general nature of arrival processes, we can focus on the dependence among successive interarrival times in the process. As discussed in [Sriram], the *index of dispersion for intervals* (IDI) can provide a broad insight into this issue. To define it, consider $\{X_k, k \geq 1\}$ as the sequence of interarrival times for the process in question. Then the IDI is the sequence $\{c_k^2, k \geq 1\}$ given by:

$$c_k^2 = \frac{k \cdot Var(S_k)}{[E(S_k)]^2} = \frac{Var(S_k)}{k[E(X_1)]^2} = \frac{k \cdot cov(X_1, X_1) + 2\sum_{j=1}^{k-1}(k-j) \cdot cov(X_1, X_{1+j})}{k[E(X_1)]^2} \qquad (k \geq 1)$$

where we assume that $\{X_k, k \geq 1\}$ is stationary, that is the joint distribution function of $(X_k, X_{k+1}, \ldots, X_{k+m})$ remains independent of k for all m. S_k denotes the sum of k consecutive interarrival times, $S_k = X_1 + \ldots + X_k$. Naturally, $cov(X_i, X_j)$ is the covariance between X_i and X_j. The final IDI expression follows by the stationarity assumption of the $\{X_k\}$ process and is quite insightful. From it, one can see that the c_k^2 represents the cumulative covariance among k consecutive interarrival times normalized by the square of the mean. The factor k appears to prevent the c_k^2 from approaching zero in the limit as $k \to \infty$. The principal usefulness of the IDI statistic is its accountingfor the *cumulative* effect of the many individual covariances of the process. In the context of a statistical multiplexer, these many covariances explain the unusually large packet delays which result under heavy load conditions.

The limiting value of the IDI, that is $c_\infty^2 = \lim_{k \to \infty} c_k^2$ completely characterizes the effect of a general stationary arrival process on a multiserver FIFO queue in heavy traffic conditions [6]. Specifically, we consider the case of a single-server queue with either finite or infinite capacity operating by a FIFO rule. The traffic input to the queue consists of the merged aggregate of a variable number of independent voice processes mentioned in the previous section. The service distributions are independent of the arrival process and assumed to be i.i.d. with a general distribution.

The IDI limit and average arrival rate of the input process serve well to model the queue's incoming traffic. There are a few basic properties of the IDI measure which can be pointed out:

- For a Poisson process, $c_k^2 = 1$ for all k
- For a renewal process, $c_k^2 = c_1^2$ for all k
- If $c_k^2 = c_1^2$ for all k, then $cov(X_i, X_j) = 0$ for all $i, j \ (i \neq j)$.

Thus, observing the fluctuations of the sequence $\{c_k^2\}$ can give intuition of the degree of deviation a process may make with a simple renewal process. Again, the covariances between process interarrivals accounts for the behavior of the IDI sequence. The third property above is the simple case in which these interarrivals are completely uncorrelated, hence equating to a renewal process. Now if we denote the index of dispersion for a superposition of n i.i.d. arrival processes by c_{kn}^2, important insights follow by respectively allowing first k to approach infinity with n fixed and then n with k fixed.

- For a superposition of n i.i.d. renewal processes, $\lim_{k \to \infty} c_{kn}^2 = c_{\infty n}^2 = c_{11}^2$ for n fixed, where c_{11}^2 is the IDI of a single interval in one of the multiplexed renewal processes.
- The superposition of n i.i.d. renewal processes, each with individual rate λ / n, tends to a Poisson process with rate λ as $n \to \infty$.

Interestingly, the latter fact also holds for the case of multiplexed non-renewal processes. However, in this case, an important condition is for the component processes to become increasingly sparse as n gets larger. We might expect that a multiplexer would see a Poisson-like aggregate arrival process if many voice sources are merged together, but what one generally observes is that great packet delays are experienced far and beyond what would be seen for a Poisson arrival process. The deviation from Poisson behavior actually increases as the number of sources increases. So the "sparsity" condition is necessary to ensure that the *total* average arrival rate remains unchanged. Clearly, in the case of this application, the voice streams are fixed in their individual average packet rates, independent of n. So it is inaccurate to always consider the superposition of the voice streams as being approximate to a Poisson process. What is observed in practice is that the Poisson approximation does work well under light traffic conditions. When higher loads exist, though, the approximation grossly underestimates packet delay since in these circumstances, the long-term covariances have a more dramatic effect. In sum, the suitability of approximating a superposition process by a Poisson process depends not only on the number of multiplexed sources but on notion of time scale. Over short intervals of time, the Poisson process does reasonably well to model the superposition process. But over longer intervals of time, the covariances come into play so that the superposition process significanly deviates from a Poisson process and displays high variability.

To actually experiment with graphing the IDI $\{c_k^2\}$ sequence, we can collect interarrival time statistics over some period of time. Figure 16 illustrates how a simple system was built to generate a sequence of these for k ranging over some selected interval. For each k, the interarrival times were collected in non-overlapping blocks of size k. Recall from the definition given previously that $c_k^2 = (k \cdot Var(S_k)) / [E(S_k)]^2$. $E(S_k)$ and $Var(S_k)$ can be straightforwardly calculated. Then for each k, we can consider the lengths of the respective blocks, denoting by Y_{ki} the length of the ith block (i.e., $Y_{k1} = X_1 + X_2 + \ldots + X_k$, $Y_{k2} = X_{k+1} + X_{k+2} + \ldots + X_{2k}$, etc.) Allow n_k to be the total number of blocks accumulated. We easily have:

$$E(S_k) = n_k^{-1} \cdot \sum_{i=1}^{n_k} Y_{ki}$$

And an estimate of the variance of a k-block would be:

$$Var(S_k) = n_k^{-1} \cdot \sum_{i=1}^{n_k} Y_{ki}^2 - \left(n_k^{-1} \cdot \sum_{i=1}^{n_k} Y_{ki} \right)^2$$

In the figure, naturally, the system is modeled in the timed DE domain. Multiple instances of Voice stars have their output streams chronologically merged together by the Merge star which sends this aggregate process to an IDI star. The IDI star is responsible for the body of the calculations. Once it is triggered on its “demand” input, it sends on its output terminals a series of ordered pairs in the form of $\left(k, c_k^2\right)$ for, in this case, $k = 1, 2, \ldots, 2000$. These data pairs are accepted by the SDF wormhole, consisting of a single Printer star which merely writes the results to a file. Recall that by the semantics of SDF operation, the wormhole will require one particle on each of its two inputs in order to fire once. Thus, each $\left(k, c_k^2\right)$ pair successively fires the wormhole and is written to to the file. The system

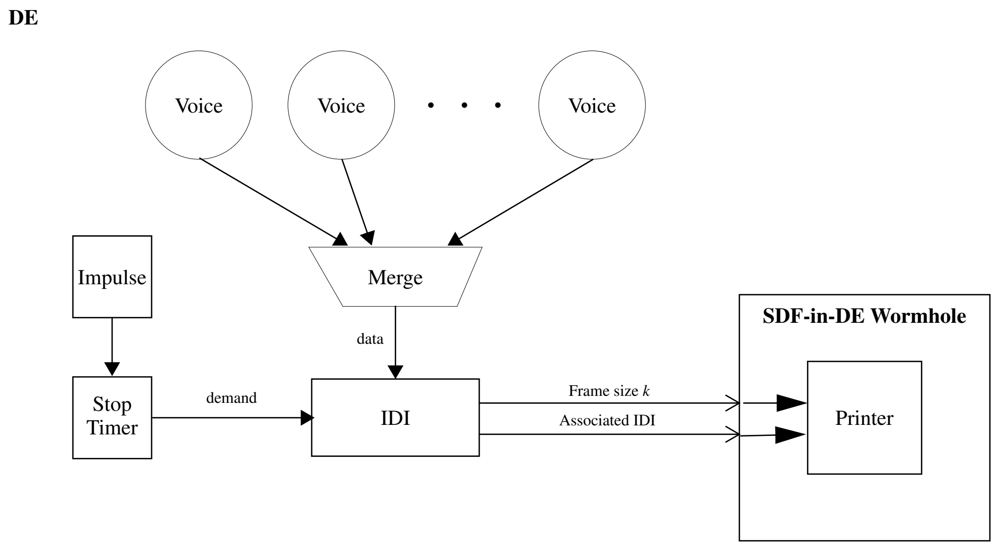


Figure 16 System to calculate IDI statistics for a superposition of voice sources

is set by the user to run for some length of time. At time zero, the Impulse star feeds a particle to the StopTimer star. The StopTimer star, having been triggered, will produce a particle at the time the simulation ends to trigger the demand input of the IDI star and have it output its statistics.

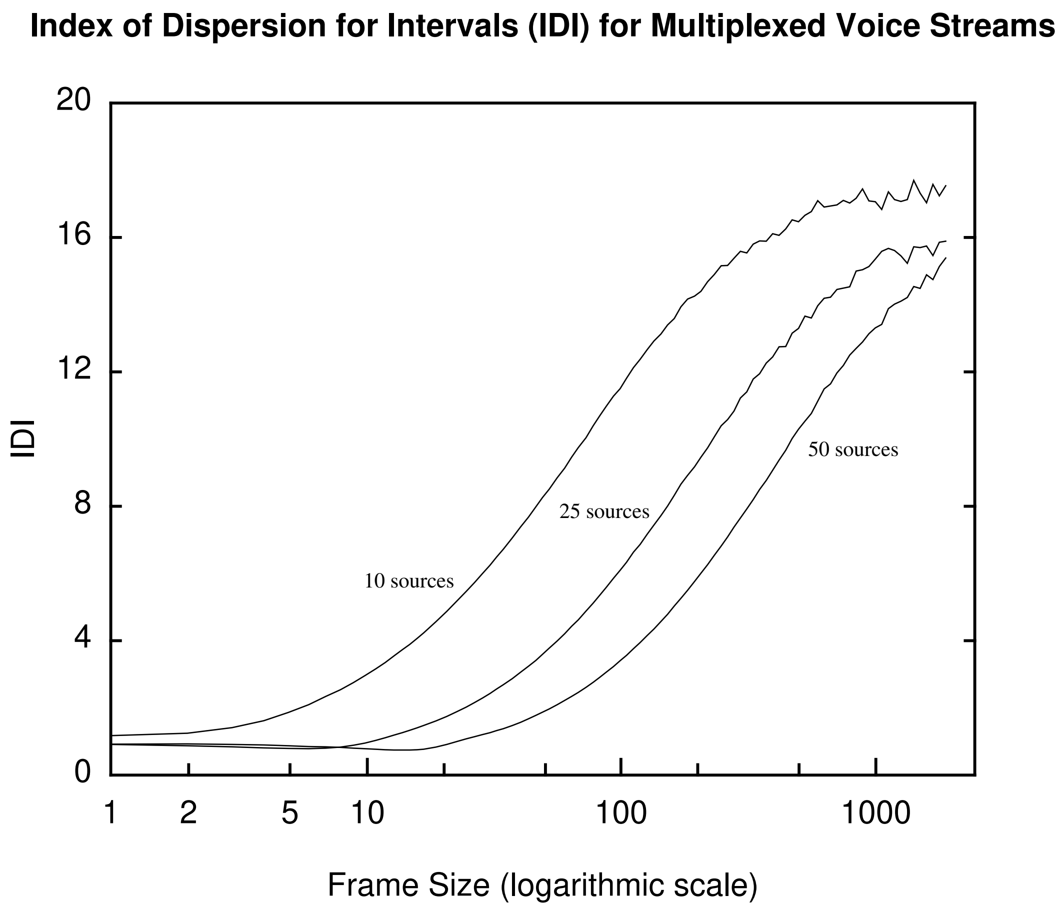


Figure 17 IDI measurements obtained by experiment

Figure 17 above illustrates the results of our IDI measurement experiment. We simulated with different numbers of multiplexed voice sources and have displayed output for the cases of $n = 10$, 25, and 50. For increasing values of n, the curve approaches in the limit a horizontal line, demonstrating that $c_{kn}^2 \to 1$ as $n \to \infty$ for all k. In other words, a superposition of infinitely many renewal processes approaches a Poisson process which we know has the property that $c_k^2 = 1$ for all k. We also observe that for a fixed n, the accumulation of interval covariances becomes increasingly significant as shown by the rising portions of the plotted curves. This qualitatively suggests the dramatic deviations from Poisson behavior of these aggregate processes as we consider large frame sizes, or significant lengths of time in which many covariances interact. Note that for small frame size values, the curves are nearly horizontal. Over these small time intervals, then, we may consider interarrivals as roughly independent.

6.0 Switch Performance Experiments

In this section, we describe a number of experiments to compare the performances of certain switching strategies. They are compared in terms of average throughput and end-to-end traversal times of packets. For each switching strategy, two basic network configurations are simulated. In the first case, a simple one-switch network is constructed without any network control element. In the second case, the entire ATM backbone network configuration of Figure 10 is implemented with the four switches operating according to the switching strategy of interest. Unlike the first case, an MQ control element is present to handle call processing functions.

6.1 Intermediate Queueing

The queueing policy pursued in the modeling of the Banyan switch in Figure 11 was that of intermediate queueing. In other words, a buffer of some finite capacity was placed at each of the two inputs to every switching element in the Banyan switch. Recall that if a packet has a destination port address represented by $(b_1 b_2 \ldots b_k)$, and has reached stage s of the switch, it should be routed to the upper output if $b_s = 0$ and to the lower output if $b_s = 1$. Packets arriving at the input(s) to a 2 x 2 switching element are inserted at the tail(s) of the corresponding queue(s). When there is a routing conflict caused by two packets contending for the same output port, the winner is randomly selected with a fair coin flip, and the loser is kept at the head of its queue, free to contend for its destination port again at the next clock cycle.

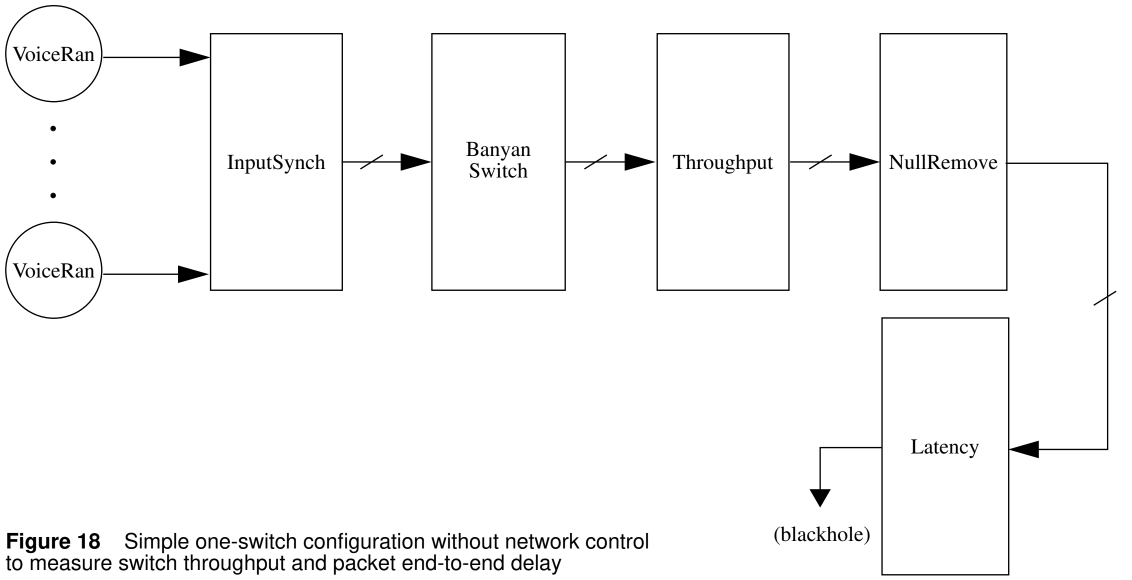


Figure 18 Simple one-switch configuration without network control to measure switch throughput and packet end-to-end delay

Figure 18 shows the first of two network configurations to be examined utilizing the policy of intermediate queueing. The external environment is modeled in the DE domain with the Banyan network built in the SDF domain as described before. As already explained, the Banyan network is a blocking network which cannot guarantee that inputs destined for distinct outputs will pass through without internal conflicts. Multiple, parallel instances of "VoiceRan" stars transmit packets according to the voice process of a single caller, and each packet is randomly routed to an output port of the switch, each with equal probability. The DEInputSynch and DENullRemove stars are placed at the inputs and the outputs to the switch to ensure smooth interaction between the SDF and DE domains as explained in section 3.2. The two stars of interest used to gather performance statistics of this network are the DEThroughput and DELatency stars.

A switch's throughput is defined as the average number of packets produced at one of the switch's output ports per clock cycle. Thus, it is a positive real number less than one. It was convenient in the case of this experiment to place the throughput measurement star at the output to the switch. For each clock cycle, each switch output port produces either an actual packet or a null packet, signifying that the port is idle for that cycle. Then it becomes straightforward to implement the star to measure the switch's throughput. The star's schematic appears as in Figure 19.

In practice, a token is generated at the end of the simulation to trigger the "demand" port of the DEThroughput star. This is done by a combination of the DEImpulse and DEStopTimer stars as shown in Figure 16. When the demand signal is detected, the star produces a set of values corresponding to the measured throughput on each of the switch's trunk lines. A packet received on one of the "input" ports is passed through directly to its respective "output" port while the star updates its internal data to reflect measured throughput.

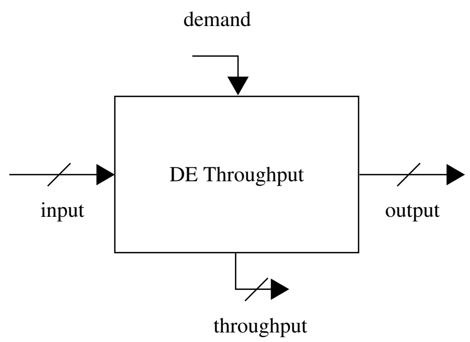


Figure 19 Schematic of DE Throughput star

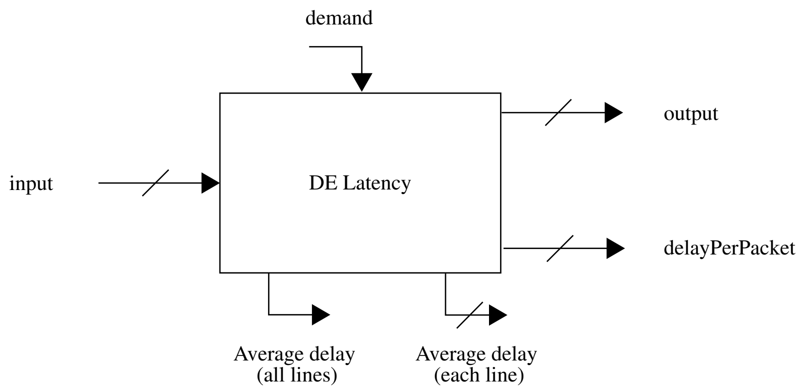


Figure 20 Schematic of DE Latency star

The other star used in Figure 18 to gather statistics about the switch's performance is the DELatency star. It follows the DENullRemove star and produces on a per-packet basis individual latency figures. This is measured by the time elapsed from the time the packet was produced by the DEVoiceRan star to its arrival at the DELatency star. Also, when triggered on the "demand" port, the star yields a set of statistics measuring the average per-packet latency time for each of the switch's trunk lines and average per-packet latency over all trunk lines considered as an aggregate whole. A block diagram of it is in Figure 20.

6.2 Input Queueing

The input queueing switching strategy consists of an input queueing structure followed by a *nonblocking* network. Each trunk line into the input queueing network has a dedicated first-in, first-out queue. On each clock cycle, the head elements of the queues contend for their destinations. Packets that request identical destinations require the random selection of one to continue through the switching network. The others that lose contention for that destination remain in their queues to contend again at the next clock cycle along with the new head elements of the other queues. Thus, packets presented to the nonblocking switch all have distinct destinations and can pass through the network without risk of conflict.

Input queueing suffers from the problem of head-of-line (HOL) blocking. Packets may remain in queue behind head elements that fail to gain admission to the switching network due to conflicts with other head elements. This represents a decrease in maximum throughput attainable by the switch. Different variants on the simple input queueing policy described here exist. We may introduce multiple priority levels among packets. In this case, though, fairness to the individual queues is an issue since lower-priority packets may suffer inordinate delay in winning contention for the switch. Also, when packets vie for a destination port, it may be sensible to adopt a "longest-queue" policy that will grant access to the packet coming from the queue with the largest number of buffered elements.

Figure 21 presents a top-level view of the different modeling approach used to implement this means of queueing as opposed to intermediate queueing from a Ptolemy standpoint. Instead of a simple DEInputSynch star, a much more complex system is built in the form of a DEInputQueue galaxy. The internals of this galaxy are presented later. Also, instead of a blocking Banyan switch, a nonblocking Batcher-Banyan network is built. Both employ a DENull-Remove star to filter out null packets offered by the switch built in SDF.

Now consider the low-level functional implementation of the Batcher-Banyan switch. At each clock cycle, a number of packets may be offered to the inputs of the Batcher network with the other inputs occupied by null packets. The purpose of the Batcher network is to sort the packets by destination address and to concentrate them toward the

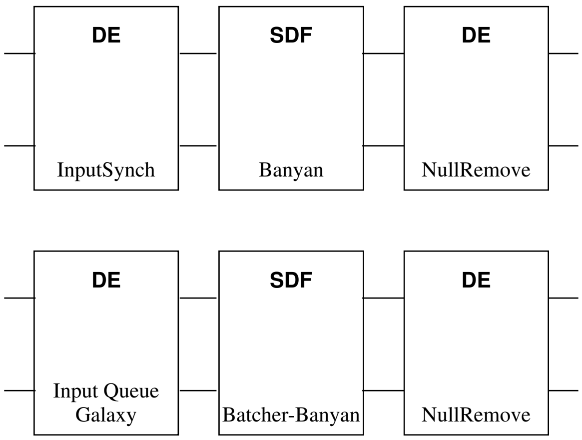


Figure 21 Top-level comparison of intermediate queueing (top) and input queueing (bottom)

top so the null packets get filtered to the bottom inputs. When this is done, packets may be sent to the Banyan network in two phases in order to avoid conflict. Each phase involves a separate, entire execution of the Banyan network. By this, it turns out for the Omega Banyan network design that a simple rule to accomplish this is to simply ensure that for a given phase, no two data packets received by any 2 x 2 switching element in the first stage of the Banyan network have the same most significant bit in the binary representations of their destination addresses. Note that this simple rule is dependent on the packets having been sorted and "concentrated" before being offered to the Banyan network.

This overall switching operation conforms quite straightforwardly to the semantics of the SDF domain. Each 2 x 2 element of the Batcher and Banyan networks accepts a single token on each of its inputs and produces a single token on both of its outputs per execution. If a Banyan network element has no data packet to produce on an output port, it yields a null packet to obey SDF semantics. However, extra stars need to be included after the Batcher network and before the Banyan network to offer the latter packets in two phases according to the previously discussed rule. Figure 22 provides an example of a pre-banyan element accepting a pair of packets, and offering them to the Banyan network in separate phases because of their identical most significant bits. Note that a "pre-Banyan" star accepts one token on each input and produces two tokens on each output per execution. As for a Banyan 2 x 2 star, it has no intermediate buffering of any kind. Any detected conflict between two data packets would represent a fatal error.

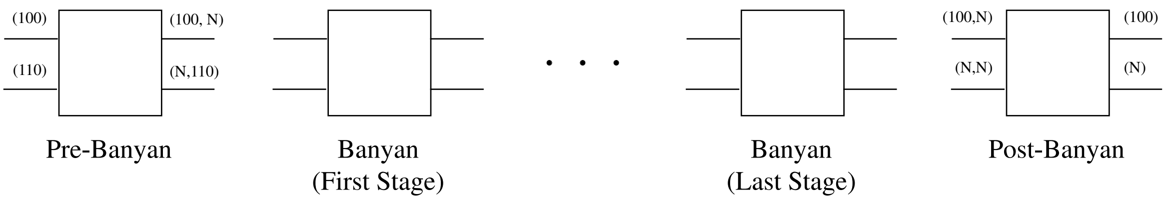


Figure 22 Low-level implementation of the Banyan network. The Pre-Banyan network produces two phases of packets for the Banyan network. Packets at a Pre-Banyan 2x2 with same MSB in output address must go in different phases to avoid conflict. The Post-Banyan stars reduce the pairs of packets produced by the Banyan network for each port to single packets to condense the two phases of operation "back into one." ("N" means null)

After the Banyan network has executed twice to account for both phases of its switching operation, it has produced a pair of packets on each output of its 2 x 2 elements in the last stage. Hence, a set of stars is necessary to condense each pair to a single packet. Of course, packets that were originally offered at the switch's inputs had no duplication of destination address so at least one packet in each of the output pairs would be null. The mechanics of the "post-Banyan" star are given in the same figure. Hence, we see that the overall modeling of the Batcher-Banyan network is quite natural when built in SDF.

The function of the DE Input Queue Galaxy is to present packets to the nonblocking Batcher-Banyan network that have no duplication of output address, queueing packets as necessary at the switch inputs and randomly resolving output destination conflicts with equal fairness to contending packets. Figure 23 structurally illustrates the stars that compose this galaxy. Of course, each star is modeled in the DE domain. In order to understand the galaxy's operation, it is necessary to describe the actions taken by each star when it is triggered into execution. After doing this, we can understand the mechanics of the galaxy as a single composite block.

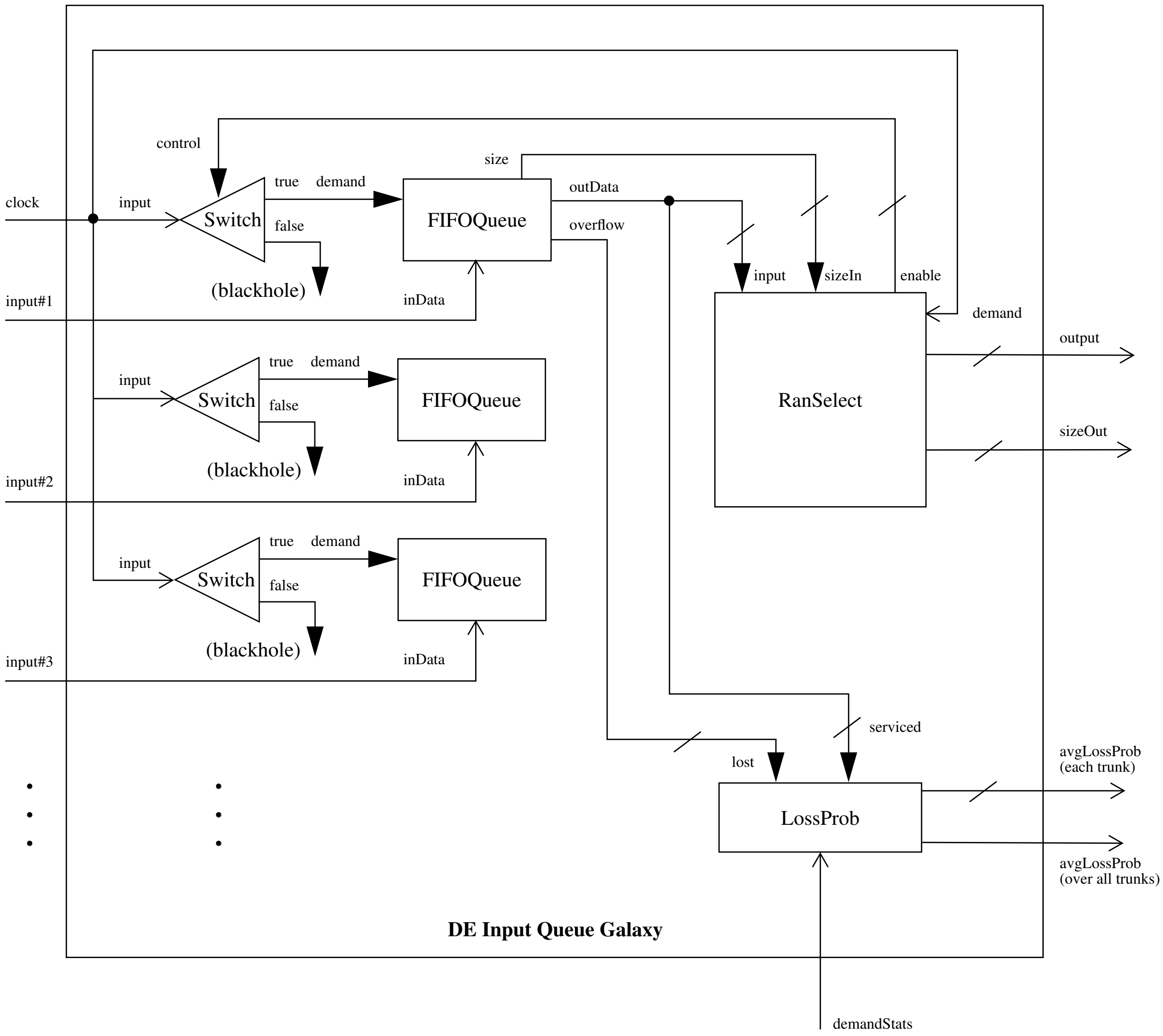


Figure 23 The design of the DE Input Queue Galaxy. There are multiple, parallel instances of Switch and FIFOQueue stars, one for each trunk line, and single instances of the LossProb and RanSelect stars. For convenience, connections to the LossProb and RanSelect stars are only shown for one trunk line. Multiportholes on these stars have their arcs marked with slashes. Connections made to stars external to the galaxy use different arrowheads than those made among stars internal to the galaxy.

The Switch star acts as a binary router. When it receives a token on its “input” port, it routes it to either its “true” or “false” output based on its most recently received input on the “control” port. As can be seen, a clock signal external to the galaxy feeds into the Switch star and can either continue onward to provide a demand signal to the FIFOQueue or to be thrown away into the BlackHole star which merely accepts tokens and disposes of them. The RanSelect star has an “enable” multiporthole which acts as the input to the Switch stars’ “control” inputs. Thus, when the RanSelect star enables a Switch star, it allows the clock signal into that Switch star to trigger that trunk line’s FIFOQueue; otherwise, by disabling it, the clock signal is shut off from the FIFOQueue.

The FIFOQueue star models a queue with either finite or infinite capacity. It is possible for the user to set a galaxy parameter that establishes the capacity of the queue instances in the galaxy or declares them to be of unbounded capacity. Each trunk line into the galaxy connects to the “inData” port of a FIFOQueue instance dedicated to that channel. If the queue has space for an incoming packet, it is inserted at the tail of the queue; otherwise, it is immediately output on the “overflow” port. When a token is received on the “demand” port, the head element of the queue, if any, is issued on the “outData” port. Note that any packet produced on the “outData” port is sent to both the “input” port of the RanSelect star and the “serviced” port of the LossProb star. Lastly, for any token received on either the “inData” or “demand” ports, after appropiate actions as mentioned above have been taken, the current occupancy level of the queue is output on the “size” port as a nonnegative integer.

There are parallel, multiple instances of the Switch and FIFOQueue stars for each incoming trunk line into the Input Queue Galaxy. On the other hand, there is a single instance of the LossProb and RanSelect stars. The function of the LossProb star is straightforward: the “outData” and “overflow” ports of each FIFOQueue star feed into it so it can yield, when triggered by a “demandStats” signal external to the galaxy, the average loss probability over each trunk line and over all lines as an aggregate whole. The RanSelect star maintains the head-of-line packet for each trunk line’s queue. When it receives a packet on its “input” mulitporthole, it records it as the head-of-line packet for that channel’s queue and sends a disable command to the corresponding Switch star so that demand signals cannot cause another packet from the queue to come forth and “collide” with the current head-of-line packet. The same demand signal that feeds into the Switch stars of the galaxy also acts as a demand signal for the RanSelect star. When it is triggered on this port, the star examines its head-of-line packets for their destinations. It chooses a set which can be safely sent according to previous discussion and for those channels that are transmitting packets, it sends an enable signal to their Switch stars in addition to transmitting the packets on “output”. For channels that lose their bids for transmission or have no packets at all, null packets are produced on “output.” Thus, this galaxy can suitably interact with the Batcher-Banyan network built in the SDF domain. When the RanSelect star receives a token on its “sizeIn” multiporthole which is connected to the “size” output ports of the FIFOQueue stars, it reproduces that number on the matching port of its “sizeOut” multiporthole. However, if there is a head-of-line packet for that channel, then it produces that number plus one.

6.3 Simulation Results

The simple one-switch network in Figure 18 was simulated for both the intermediate and input queueing policy mechanisms. Recall the structures of the switches and their interfaces to the external network which have already been discussed and diagrammed from a top-level point of view in Figure 21. Single voice process generators transmit randomly to switch outputs, and in both cases, the time slot interval used was that of the packetization period of voice traffic, 16 ms. Results are given in Figure 24. As can be seen, greater average switch throughput was achieved with

		Input			
	Intermediate	5	10	15	20
Avg Tput	.3662	.3371	.3492	.3542	.3564
Avg Latency	.0777	.0591	.0821	.0946	.1008

Figure 24 Average throughput and latency figures for Figure 18 under intermediate queueing and input queueing with different input buffer capacities.

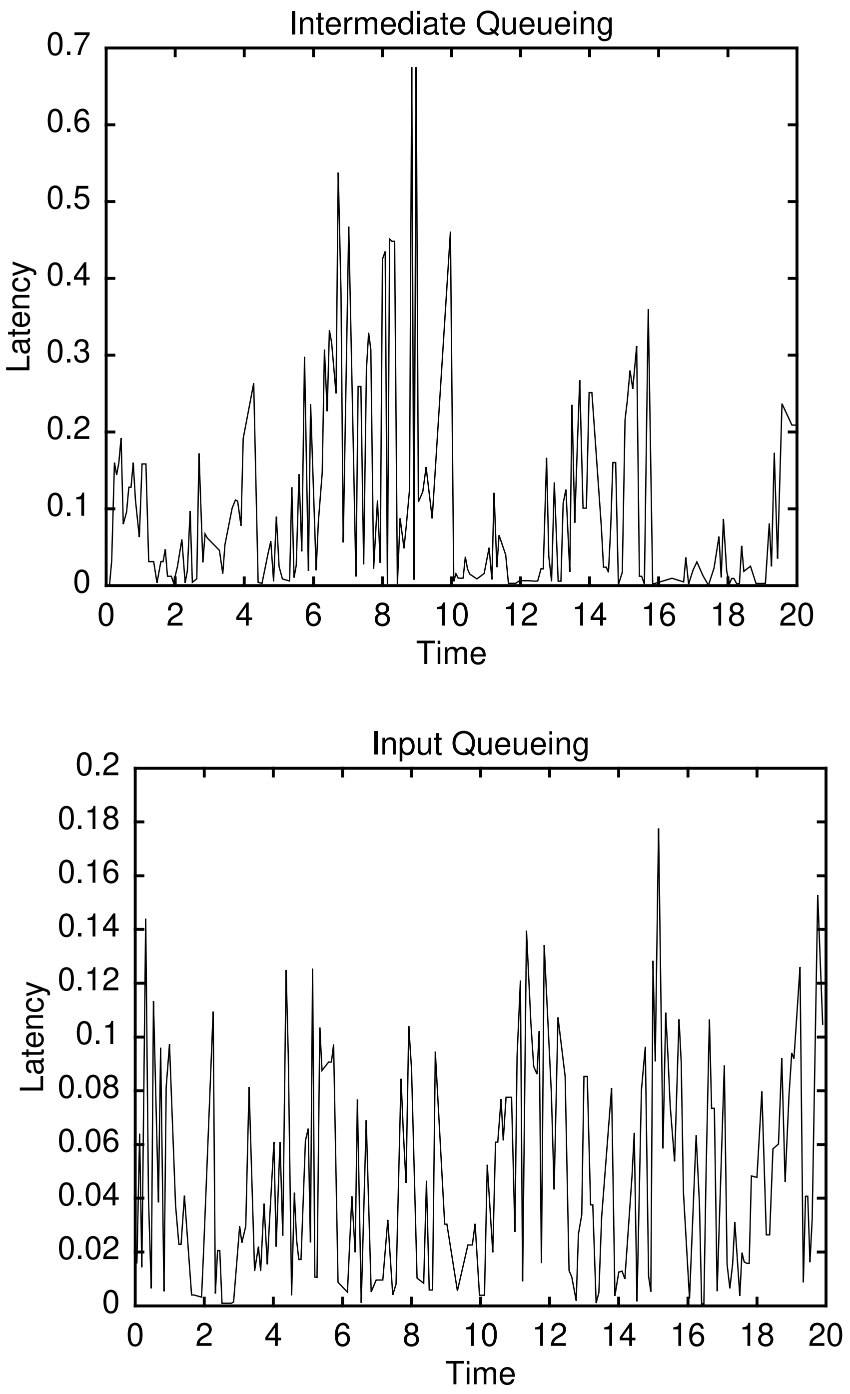


Figure 25 Per-packet latency statistics for intermediate queueing and input queueing (queue capacity for input queueing was set at five.) The graphs monitor statistics for twenty simulated time units.

the intermediate queueing policy. The sizes of the intermediate queues used in the 2 x 2 elements of the Banyan network were sufficiently large to prevent packet loss; however, queues used in the input queueing policy were experimented with at different capacities. As the capacity was increased, average switch throughput level increased as well. Gains made in throughput level taper off as queue capacity is allowed to increase in accordance with the concept of diminishing returns. In the case of this simulation, average latency for intermediate queueing was appoximately five times the length of a time slot. Average loss probabilities for the input queueing policy decay at a rate faster than that of a linear one with respect to queue capacity. Figure 25 graphs per-packet latency statistics for the two queueing policies in this configuration for one of the randomly selected switch lines. Intermediate queueing clearly displays a higher degree of variability than input queueing, and this observation is intutively reasonable. The graphs merely display statistics for the first twenty simulated time units. In just this time interval, though, the curves reaches a peak of approximately .67 seconds in the intermediate queueing graph as opposed to a mere .18 seconds in the input queueing graph.

Figure 26 shows an approximate scatter plot that monitors input queueing occupancy level for the first twenty units of simulated time for one of the selected input buffers.

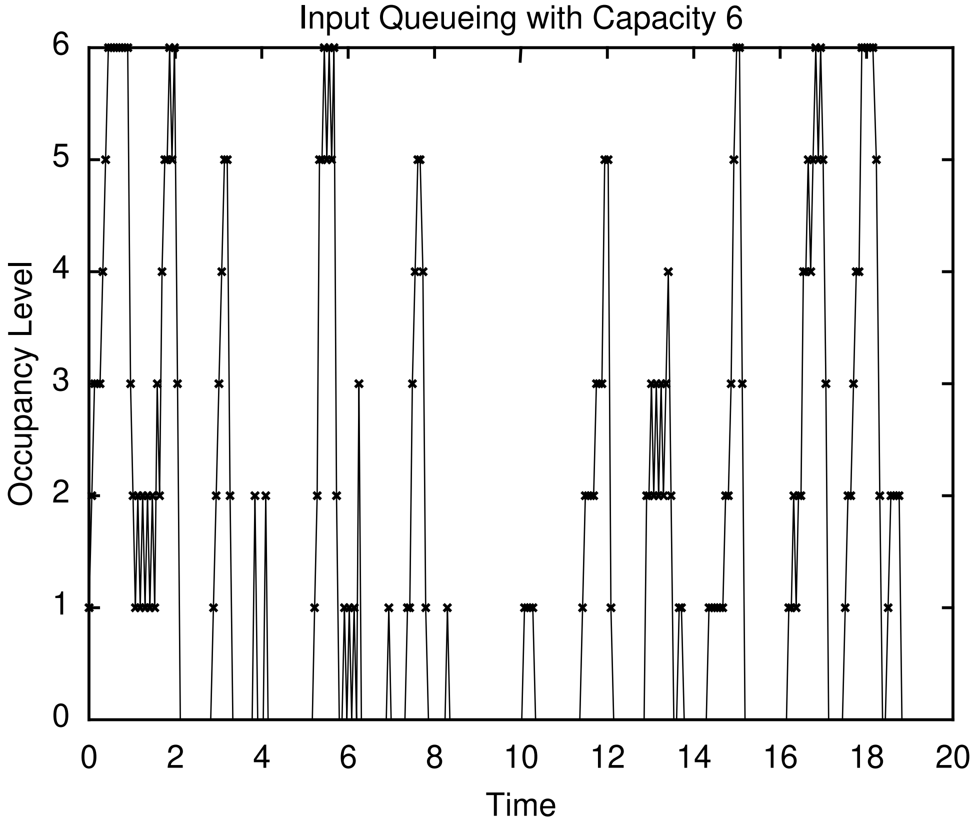


Figure 26 Input queue occupancy level for twenty time units

Next, the four-switch network of Figure 10 with network control was simulated for the intermediate and input queueing policies under identical conditions. The switches are structured as described in the previous one-switch network scenario. Figure 10 shows how the four switches are numerically designated. As for trunk lines, each switch has eight of them. The first three, numbers one through three, are connected with customer nodes. Trunk lines four and five are drawn vertically, six diagonally, and seven and eight horizontally. These simulations follow the description of section 3.1 and allow a maximum of one conversation to be supported over a given trunk line. As above, customer nodes are represented by single voice source processes and the time slot interval of the switches is set at 16 ms. Results in this case were much different as Figure 27 indicates. The latency figures correspond to traversal time of packets routed to the twelve customer nodes in the network, three for each of the four switches. Statistics for average per-packet latency and switch throughput per trunk line are given. Intermediate queueing, although achieving higher average switch throughput, suffers greatly in average per-packet latency time. Trunk lines four through eight can be considered as the “inter-switch” trunk lines and among those, trunk line six averages the highest throughput, naturally, because switches diagonally opposite each other are singly connected instead of diagonally connected. Thus, it would more frequently support traffic in a randomly established conversation.

LATENCY

		Intermediate				Input			
		SW1	SW2	SW3	SW 4	SW 1	SW 2	SW 3	SW 4
Trunk Line No.	1	.2686	.3225	.3329	.2671	.0259	.0235	.0241	.0269
	2	.2553	.3009	.3357	.2824	.0263	.0256	.0272	.0210
	3	.2921	.2711	.2863	.2733	.0293	.0280	.0251	.0258

THROUGHPUT

		Intermediate				Input			
		SW1	SW2	SW3	SW 4	SW 1	SW 2	SW 3	SW 4
Trunk Line No.	1	.2785	.2643	.2748	.2628	.1539	.1514	.1500	.1500
	2	.2613	.2620	.2804	.2698	.1340	.1339	.1464	.1305
	3	.2690	.2778	.2622	.2755	.1408	.1391	.1133	.1340
	4	.1160	.1166	.1212	.1170	.0455	.0476	.0619	.0610
	5	.1240	.1222	.1261	.1297	.0605	.0629	.0671	.0609
	6	.1949	.2125	.2054	.2019	.1249	.1133	.1144	.1046
	7	.1357	.1167	.1184	.1354	.0620	.0685	.0679	.0592
	8	.1257	.1258	.1208	.1207	.0662	.0568	.0562	.0747

Figure 27 Throughput and latency statistics for network-level simulation, intermediate and output queueing

7.0 Conclusion

We have discussed the diverse issues that are encountered in the simulation of any large-scale telecommunications network: modeling of source traffic, network control, switch design, and the effective interaction of all these elements in a unified context. The Ptolemy system was introduced to address these issues. Specifically, we considered a cell-relay network with centralized control that provided basic call-processing capabilities. The Ptolemy MQ domain was developed to support the design of this control element, and specific examples were given to illustrate its use in modeling control subsystems.

Also, we considered the design of space-division packet switches with associated queueing policies. Two basic designs were presented, that of a pure Banyan network utilizing intermediate queueing and an input queueing strucure followed by a nonblocking Batcher-Banyan cascade network. Our primary measures of performance of these switches were their average throughput and the end-to-end delay incurred by packets. We presented results of simulations to gather these statistics for two scenarios: simple-one switch configurations with "always-active" sources and a complete large-scale ATM network governed by a centralized control. Thus, by explaining the issues involved in the modeling of these network elements and the nature of their interaction, challenges that arise in the successful simulation of any large-scale, heterogeneous system become clear.

8.0 References

[1] S. Bhattachartta and E. A. Lee, Aug. 1991, "Scheduling Synchronous Dataflow Graphs for Efficient Looping," M.S. Report, Electronics Research Laboratory, Dept. of EECS, U. California Berkeley.

[2] J. Buck, S. Ha, E. A. Lee, and D. G. Messerschmitt, 1992, "Ptolemy: A Framework for Simulating and Prototyping Heterogeneous Systems", *International Journal of Computer Simulation*.

[3] M. Karol, M. Hluchyj, and S. Morgan, Dec. 1987, "Input Versus Output Queueing on a Space Division Packet Switch", *IEEE Transactions on Communications*.

[4] J. Loh, Aug. 1991, "ATM Packet Speech Simulation Using Ptolemy," M.S. Report, Electronics Research Laboratory, Dept. of EECS, U. California Berkeley.

[5] K. Sato, S. Ohta, and I. Tokizawa, Aug. 1990, "Broad-Band ATM Network Architecture Based on Virtual Paths," *IEEE Trans. Commun.*, vol. 38.

[6] K. Sriram, Sept. 1986, "Characterizing Superposition Arrival Processes in Packet Multiplexers for Voice and Data," *IEEE Journal on Selected Areas in Communications*, vol. 6.

[7] B. Stroustrup, 1991, *The C++ Programming Language* (2nd ed.), AT&T Bell Laboratories, Murray Hill, NJ.

[8] F. Tobagi, Jan. 1990, "Fast Packet Switch Architectures for Broadband Integrated Services Digital Networks", *Proceedings of the IEEE*.

[9] J. Walrand, 1991, *Communication Networks: A First Course*, Aksen Associates Incorporated Publishers, Boston, MA.