

Chapter 9

Selected Examples

This chapter shows examples of several common modeling structures. These models address such subjects as queues with reneging, priority queues, batch arrivals, and servers that break down. The examples are meant only to show how you would model these typical situations using the QSIM Application. They are not meant to show how you would analyze these models to evaluate them or identify optimal parameterizations.

Queues with Reneging

When a customer arrives at a facility that includes queues and service, they may choose to enter a queue, if there is room, or leave the facility. Once in a queue, they may choose to leave it if they have waited too long. Not entering a queue and leaving a queue are two types of reneging. This example shows how to model some typical queues with reneging.

The model in Figure 9.1 shows a single queue for three servers. It models the M/M/c/K system where $c = 3$ and $K = 50$. This system has Poisson arrivals to a single queue with a capacity of K transactions for service by c parallel servers. Another way to model this system is with the MServer, as shown in Figure 9.1.

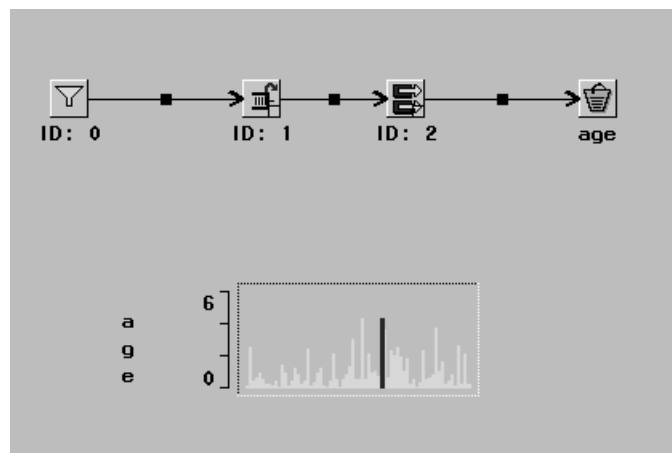


Figure 9.1. An M/M/c/K Model

This model is often compared to one with c parallel queues and servers, as shown in Figure 9.2.

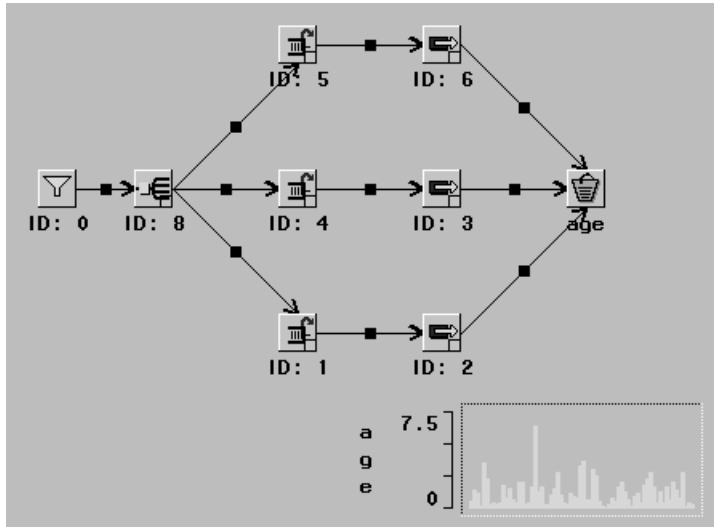


Figure 9.2. A 3-Queue 3-Server Model

In this model, the Switch component directs the transaction to one of the three queues. In this case, the transaction is routed to the shortest length queue. This is accomplished with two formulas tied to the switch. A model for this is shown in Figure 2.11.

Another variant on the parallel server models in Figure 9.1 and Figure 9.2 has customers entering a queue and, if they have waited for too long, deciding to switch to another queue. This decision-making and queue-switching policy is more complex, but it can be modeled as shown in Figure 9.3.

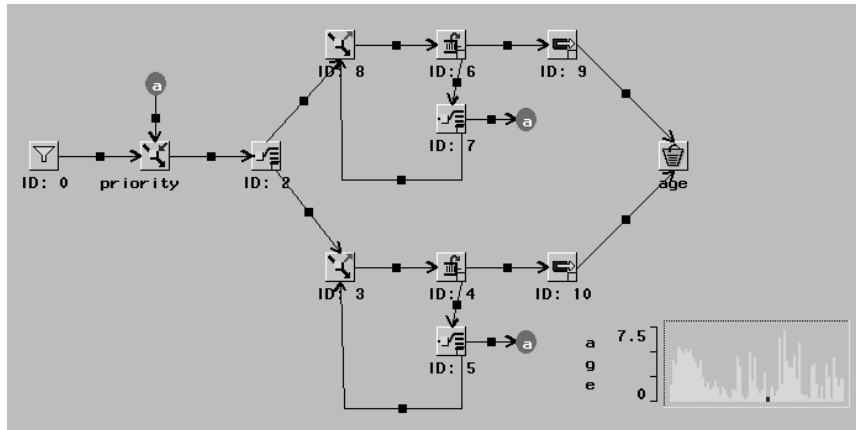


Figure 9.3. A 2-Queue 2-Server Model with Reneging

In this model, upon arrival, the transaction is assigned an attribute named “priority,” whose value is the current simulation time. This is done in the Modifier component labeled “priority.” Next, the transaction goes to a Switch, which compares the two queues and sends the transaction down the path leading to the shorter of the two queues. Next, the transaction encounters a Trigger, which schedules the transaction to balk when it has spent a given amount of time in the queue. The default is a random variable with exponential distribution with mean 1. When a transaction balks, it goes

into another Switch, which checks whether the other queue is shorter. If it is, the transaction is routed to the Connector a and goes to the end of the other queue. Otherwise, the transaction goes back into the queue after scheduling, in the Trigger, another future check.

Scanning a Queue

In the preceding example, transactions balked from a queue at a time scheduled, by a trigger, before the transaction entered the queue. Similar behavior can be modeled by periodically scanning all the transactions in a queue and balking those transactions that meet some criteria that may be based on the state of the system.

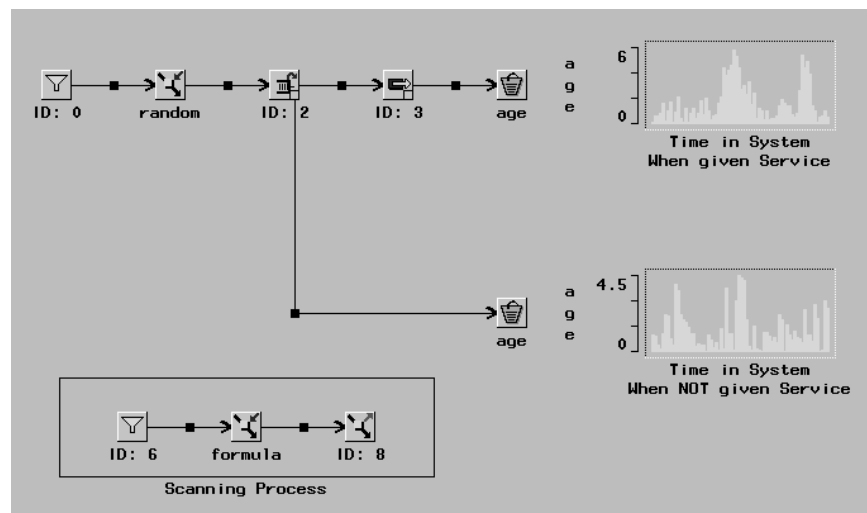



Figure 9.4. Scanning a Queue

Figure 9.4 shows a simple model where the scanning process controls the periodic searching of the queue in the main process. The sampler in the section labeled “Scanning Process” has deterministic, inter-arrival time distribution so that at fixed times a transaction gets a formula and goes to the Trigger that starts the scan of the queue.

The formula has  < where the transaction attribute is random. Since the trigger is set to the queue in the main process and the *filter* trigger message, when the transaction arrives at the Trigger, the queue is scanned and each transaction whose value of the *random* attribute is less than .8 is balked.

Priority Queues

Many types of models require that multiple classes of transactions be served by a single server. For example, two different types of customers arrive at an auto repair shop. One type needs only minor repairs, and the other type needs more major work. If one class has priority on getting service, then a priority queue is the appropriate modeling choice. Figure 9.5 shows one such model having two classes.

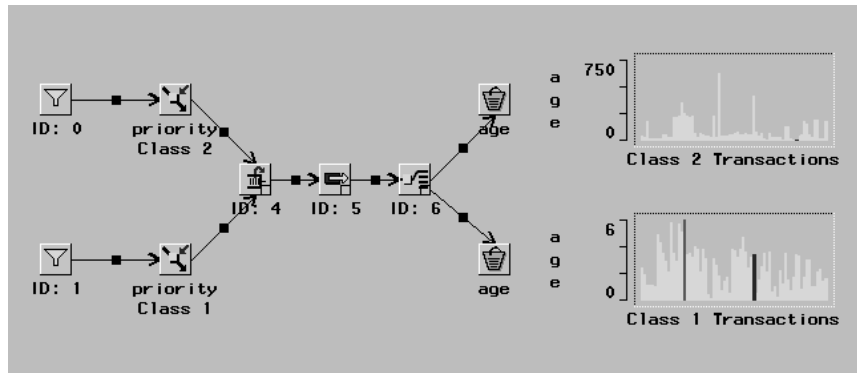


Figure 9.5. A Priority Queue Example with Two Transaction Classes

The two classes of transactions arrive according to independent Poisson processes as represented by the two Samplers. Transactions travel to Modifiers that set the priority to be either a 1 or a 2. They then enter a Priority Queue with the priority level determining their position in the queue; the higher priority transactions are serviced before the lower priority transactions. When the transactions finish service, they enter a Switch that directs them to one of two Buckets as a function of priority class.

The model in Figure 9.5 assumes that if a Class 1 transaction is in service when a Class 2 transaction arrives, the Class 1 transaction completes service before the Class 2 transaction starts service ahead of any other Class 1 transactions in the queue. Figure 9.6 shows how you would modify the model if you wanted to pre-empt a Class 1 transaction that was in service when a Class 2 arrived.

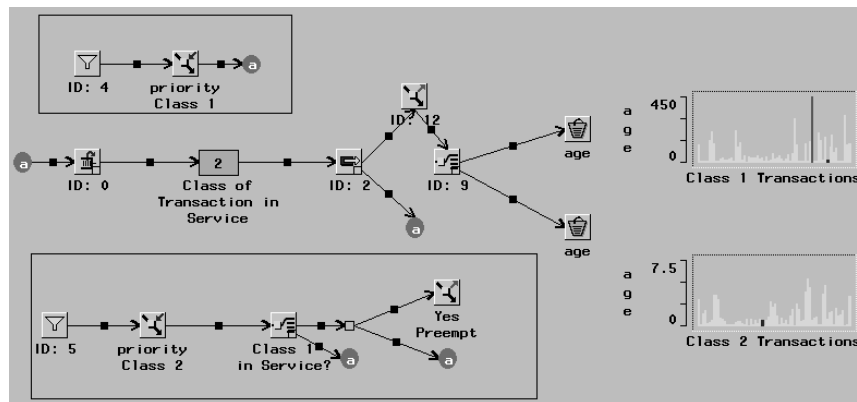


Figure 9.6. A Priority Queue Example with Two Transaction Classes and Pre-emption

For this pre-emption, you would store the class of the transaction currently in service in a number holder. This storage is done by the trigger just below the number holder. Then, when a Class 2 transaction arrives, it causes a check of the class of the transaction in service. If it is Class 1, then it is pre-empted. Notice that any pre-empted Class 1 transactions are routed back into the queue.

Batch Arrivals I

The model in Figure 9.7 shows one way to represent batch arrivals. The compound component labeled “Arrival Process” has a sampler with the batch inter-arrival time distribution set. When a transaction arrives in this process, it traverses to the Trigger labeled “Reset” which resets the Sampler labeled “Batch Source.”

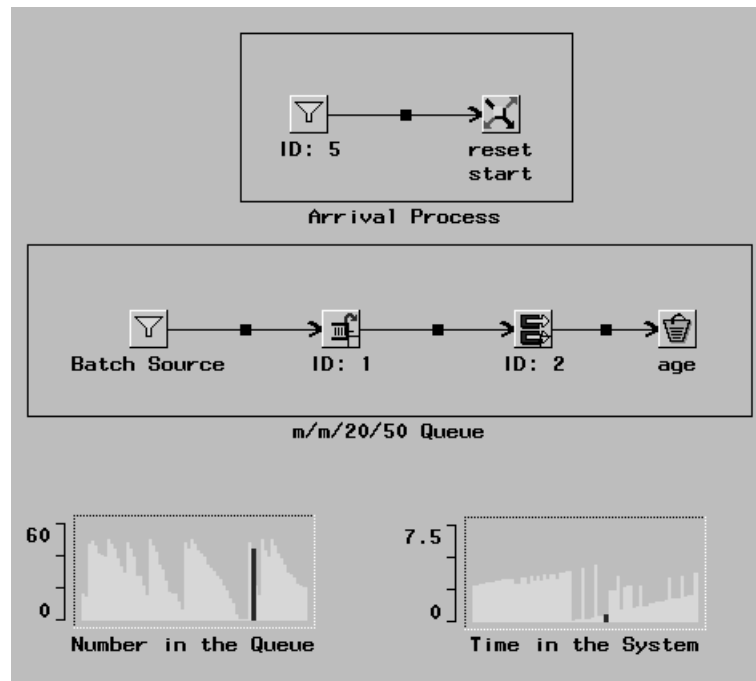


Figure 9.7. Batch Arrivals

The transaction then goes to the Trigger labeled “Start” which starts the Batch Source Sampler. The **Batch Source** Sampler has a deterministic inter-arrival time distribution with parameter 0 and capacity c , the batch size. Because of this inter-arrival time distribution, when the **Batch Source** Sampler is started by the **Start** Trigger, the **Batch Source** generates c transactions at the current simulation time and sends them to the queue. This is the batch arrival of transactions.

Notice the LinePlot labeled “Number in the Queue.”. The periodic discrete jumps in queue length show a batch arrival.

Batch Arrivals II

Another variant on batch arrivals has batch size as a random variable. A simple extension to the previous model provides this alternative.

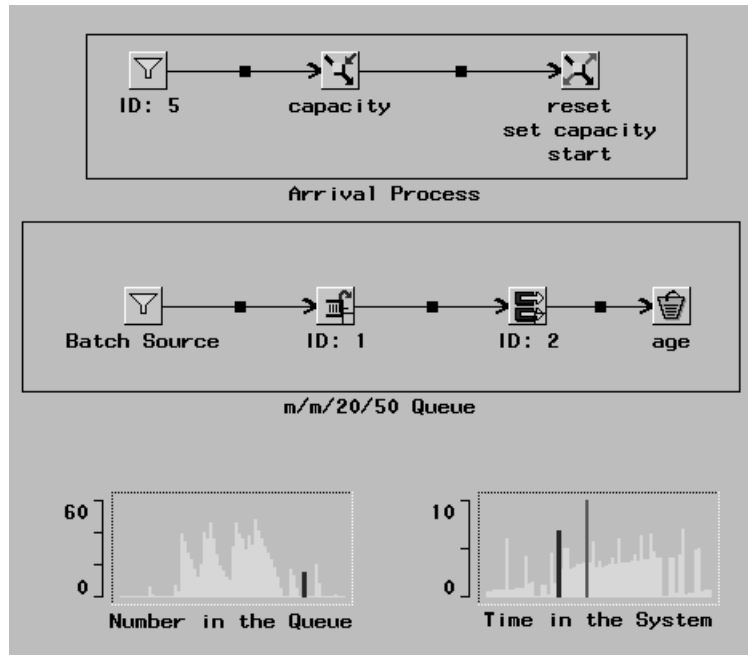


Figure 9.8. Batch Arrivals

Figure 9.8 shows two components added to the model of Figure 9.7: a Modifier labeled “capacity” and a Trigger labeled “Set Capacity.”. The capacity Modifier samples a uniform random variable on the interval $[0, 30]$ and sets it in the *capacity* attribute on the transaction in the Arrival Process. The Set Capacity Trigger then sets the capacity of the **Batch Source** Sampler to that value. Then, the **Start** Trigger starts the Batch Source arrivals as before.

Compare the LinePlot labeled “Number in the Queue” in this model to the previous example. Here, in addition to the timing of the discrete jumps in queue length, the size of the discrete jumps in queue length is random.

Nonhomogeneous Poisson Processes

In many situations, the arrival rate or service rate is determined by a Poisson process whose parameter varies as a function of time. For example, if the arrival rate to a fast food restaurant varies with the time of day and increases to a local maximum during meal times, you can sample from a nonhomogeneous Poisson process. In the QSIM Application there are some limitations to the shape of the rate function that are allowed. This function must be cyclical and bounded. The software takes the absolute value of the rate function to guarantee that it is nonnegative.

In this example, shown in Figure 9.9, the model has deterministic arrival rate and nonhomogeneous Poisson service times.

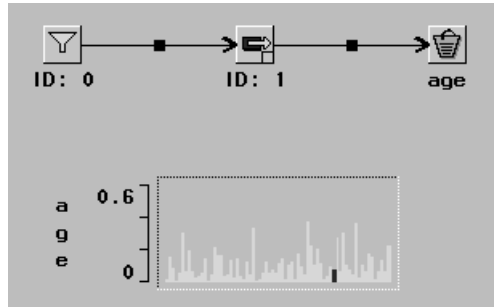


Figure 9.9. Nonhomogeneous Poisson Service

The service rate is $9 + \cos(0.001t)$, where t is the value of the simulation time when a sample is taken. This rate function is specified via the control panel for the random variable, as shown in Figure 9.10.

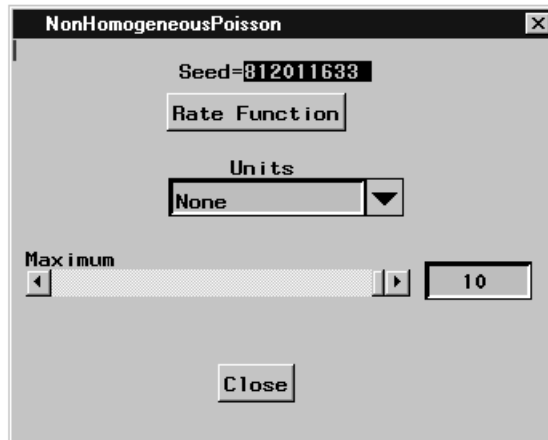


Figure 9.10. Nonhomogeneous Poisson Control Panel

In this window you can set two parameters of the process: the rate function and the maximum value that the rate function can take. The rate function is specified as a QSIM formula. When you click the **Rate Function** button, a Formula Manager window opens and allows you to specify the function.



Figure 9.11. Rate Function

Figure 9.11 shows the function used in this example. The maximum is needed by the algorithm that does the sampling. If this is not the correct maximum or the function specified is not cyclical, then the sample is not from the desired distribution.

When the transactions from this simple model are displayed in the LinePlot as shown in Figure 9.9, you can see the impact of the cyclical rate function on the transaction time in the system.

Markov-modulated Poisson Arrivals

A Markov-modulated Poisson Process (MMPP) is a Poisson process that has its parameter controlled by a Markov process. These arrival processes are typical in communications modeling where time-varying arrival rates capture some of the important correlations between inter-arrival times. This example has a Markov-modulated Poisson process that serves to control the arrival process to a single-queue, single-server queueing model.

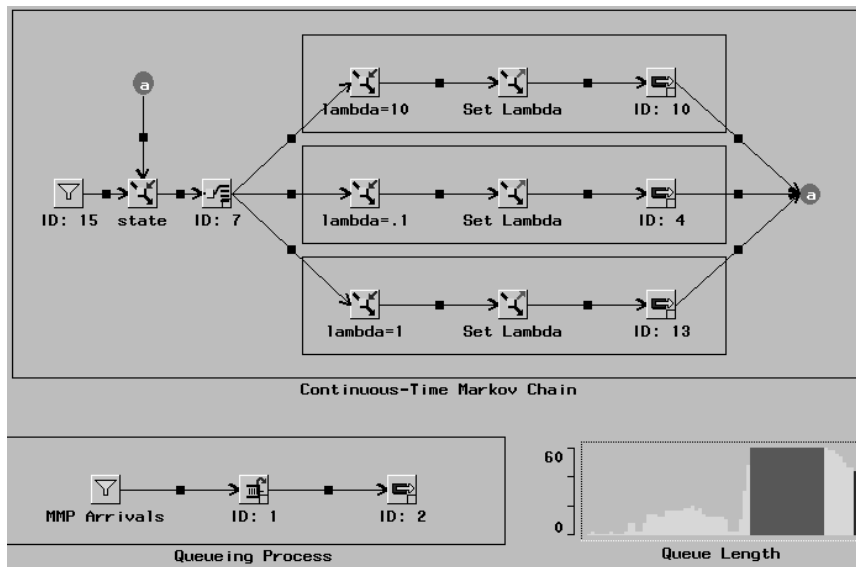


Figure 9.12. Markov-Modulated Poisson Arrivals

Figure 9.12 shows one way to model an MMPP. The process labeled “Markov-modulated Poisson Process” samples from an MMPP distribution and sets the value of the parameter λ , the mean inter-arrival time for an exponential random variable in the Sampler labeled “MMPP Arrivals.”. In the upper process, λ is given the values 10, .1, and 1 based on the state of a Markov chain. The state is changed in the Modifier components labeled “state.” Each has a conditional component driven by an observation of a uniform random variable. So, for a given state, the state is changed to the next state and the value of λ is chosen for the **MMPP Arrivals** Sampler. The selected λ is set in the **MMPP Arrivals** Sampler, and the process is delayed for an exponential amount of time whose parameter is state dependent. The transaction then goes to a switch that routes based on the state for the next state change.

State-Dependent Service

In many situations, the rate of service depends on the type of service being performed. For example, the time it takes for a teller to service a customer in a bank depends on the type of service requested. State-dependent service distributions are modeled similarly to the Markov-modulated Poisson arrivals. Consider the example shown in Figure 9.13, in which there are multiple classes of transactions to a single queue.

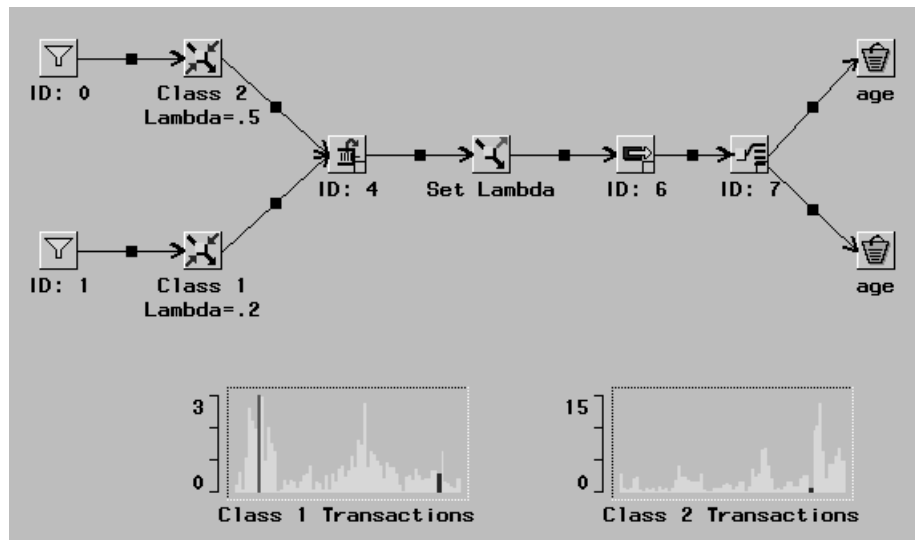


Figure 9.13. State Dependent Service

Figure 9.13 shows this model with the addition of a Modifier to set the exponential mean service time. When the transaction leaves the queue to begin service, it passes through a Trigger that sets the parameter for the service time as a function of the class of transaction that is to receive the service. In addition to the parameter, other models can change the shape of the service distribution.

Servers that Break Down I

You may have a need to model a server that periodically breaks down and is repaired. For example, a machine on an assembly line may periodically fail. Figure 9.14 shows such a model. The Server component labeled is the server that experiences down periods when it cannot service transactions.

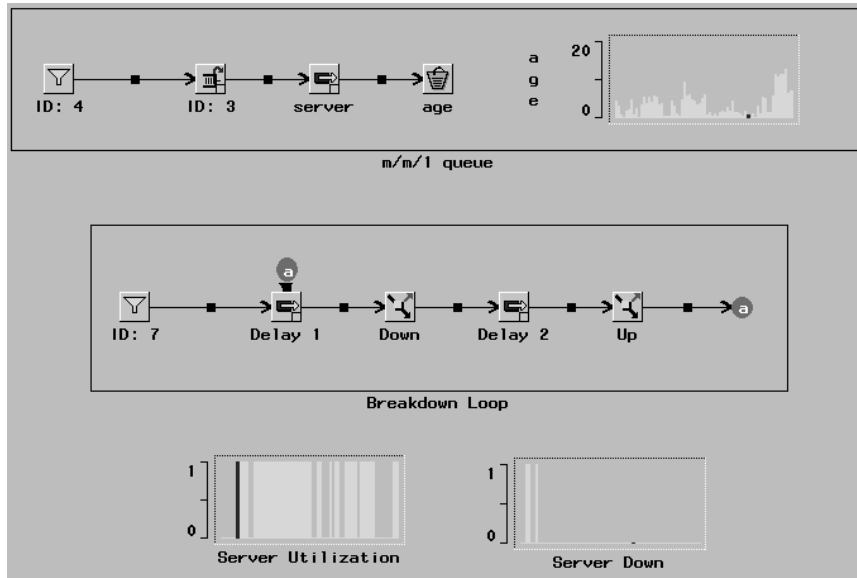


Figure 9.14. A Server that Breaks Down

The process in the compound component labeled “Breakdown Loop” models the breakdown behavior. The transaction pool has a capacity of 1 so that, when it is started, one transaction is generated that cycles through the breakdown loop for the rest of the simulation. This loop has two delays: Delay 1 models the time when the server is in operation; Delay 2 models the time when the server is broken. The two triggers, labeled “Down” and “Up,” stop and start the Server.

Servers that Break Down II

Another variant of the server breakdown model concerns what happens to the transaction that is in service when the breakdown occurs. In the model in Figure 9.14, even though the server is stopped when it breaks, the transaction in service completes service. The model in Figure 9.15 adds the pre-emption of the transaction in service, which is routed back into the queue.

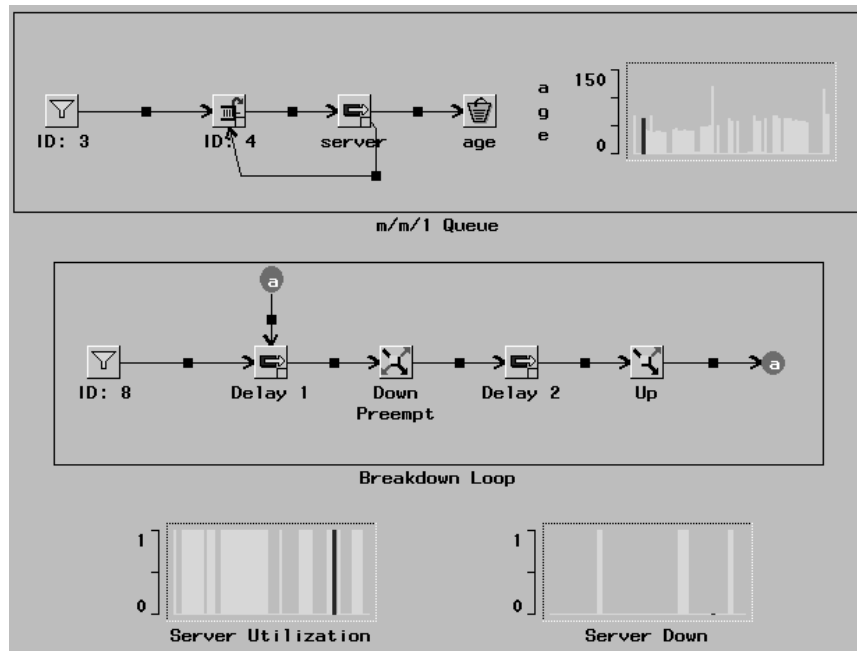


Figure 9.15. A Server that Breaks Down

By default, the transaction is placed at the end of the FIFO queue. So, if there were other transactions waiting for service, the pre-empted transaction would be behind them. Another variant on this model would place the pre-empted transaction into the front of the queue even though the queue was a FIFO for nonpre-empted transactions. This variant could be accomplished using a priority queue where the transaction priority is the simulation time at the time the transaction arrived to the queue and the queue has decreasing priority (see Figure 2.5). See the preceding example on priority queues.

Batch Service I

Suppose that you want to service transactions in a batch where you start service simultaneously on all the items in the batch but the individual service times are independent and identically distributed. This might occur in a drying process, where you have arrivals to a drying machine determined by some arrival process. When there are enough items to fill the batch, the baking of all the items in the batch begins. However, as each item dries it is removed individually from the drying machine.

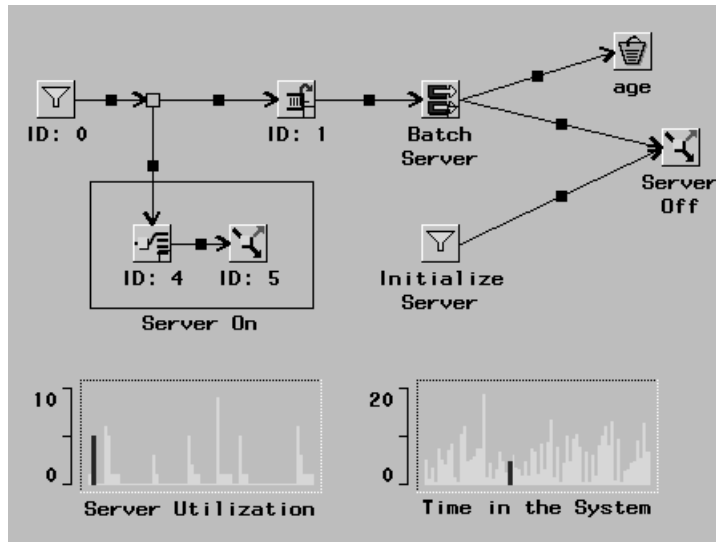


Figure 9.16. Batch Service

The model in Figure 9.16 accomplishes this batch service. In this model the multiple-server is set to the batch size and the **Server On** compound component turns the server on if it is empty and there are 10 or more transactions in the FIFOQueue. The Trigger labeled “Server Off” turns the server off when each transaction leaves service. When a server is off all transactions currently in service complete normally, but the server will not send out messages for additional transactions. As a result, service on all transactions in process will complete, but additional arrivals to the system will queue until there are at least 10 and the server is empty.

Notice the LinePlot labeled “Server Utilization.” It shows the number of transactions in service over time. It demonstrates graphically the batch service and independent nature of the service completions.

Batch Service II

Some situations demand a somewhat different approach to batch service; for example, consider a washing machine. The machine is started when enough items have arrived for service to complete the batch. However, unlike the preceding example, all the items in the batch finish at the same time. The model in Figure 9.17 accomplishes this.

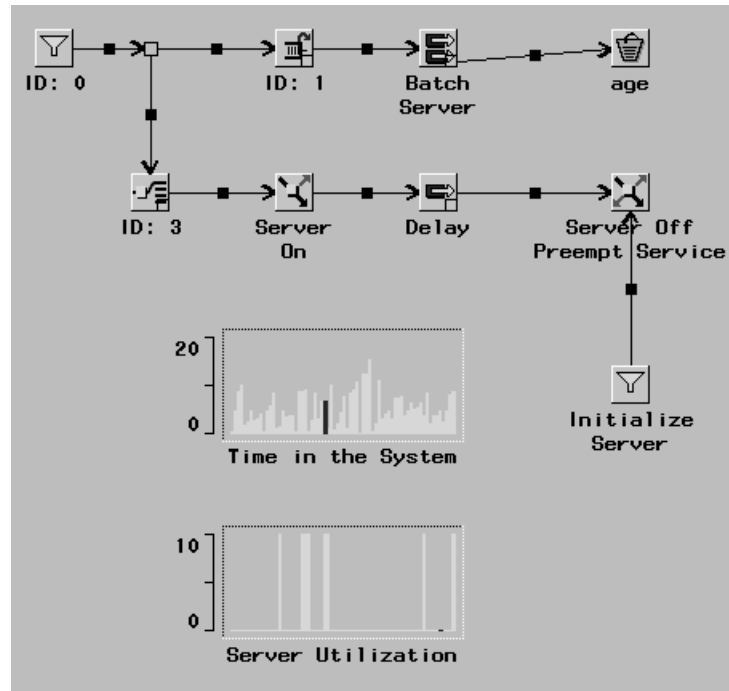


Figure 9.17. Batch Service

In this model the service distribution in the MServer labeled “Batch Server” is deterministic with a large parameter value for example, D . The Server labeled “Delay” provides the actual sample of the service time for the entire batch. And the Batch Server is turned on before the Delay and off after service for the batch is complete. Since turning the server off does not pre-empt transactions currently in service, there is another Trigger labeled “Preempt” that pre-empts all the transactions in the Batch Server. Since the transactions are pre-empted, they leave the server using the balk node.

Notice the LinePlot labeled “Server Utilization.” It shows the number of transactions in service over time. It demonstrates graphically the batch service and dependent nature of the service completions.

Because of the modeling technique used here, the service time distribution is the minimum of D and \mathcal{X} , an exponential random variable. If you want the service time distribution to be \mathcal{X} , then use caution in choosing D so that the probability that $\mathcal{X} > D$ is very small and highly unlikely to occur within the number of samples planned.

Batch Service III

Another variant on batch service has the transactions accumulating into a batch according to the arrival process and has service scheduled as soon as the first transaction arrives. Service on the entire batch completes at once.

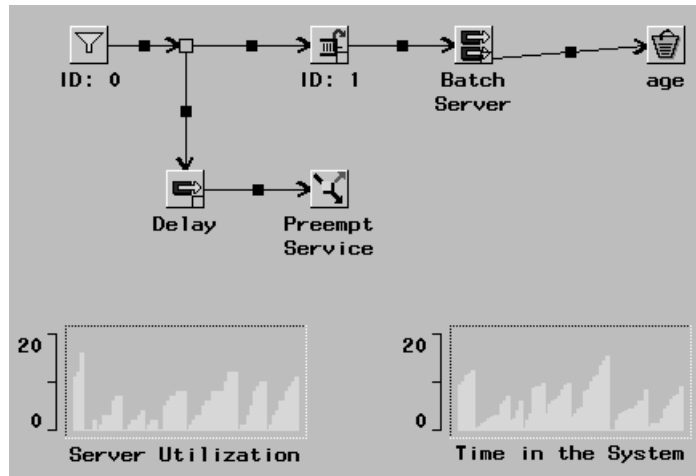


Figure 9.18. Batch Service

Figure 9.18 shows a model of this batch service. When the first transaction arrives to the Server labeled “Delay” it initiates the definition of a batch. Any other transactions that arrive to that server are discarded through the balk node. When the delay is complete, the Trigger labeled “Preempt Service” terminates service on all the transactions in the MServer labeled “Batch Server.” As the transactions arrive they accumulate in the MServer for batch service. This server has a deterministic service distribution with a large parameter value for example, D . Note that because of the modeling technique used here, the service time distribution is the minimum of D and \mathcal{X} , an exponential random variable. If you want the service time distribution to be \mathcal{X} , then use caution in choosing D so that the probability that $\mathcal{X} > D$ is very small and highly unlikely to occur within the number of samples planned.

Notice the LinePlot labeled “Server Utilization.” It shows the number of transactions in service over time. It demonstrates graphically the accumulation of the batch and the dependent nature of the service completions.

Assembly

In a model of manufacturing systems, there is often assembly of subunits into larger units. The assembly cannot occur unless all of the subunit pieces are available. An important component for modeling this behavior is the Adder.

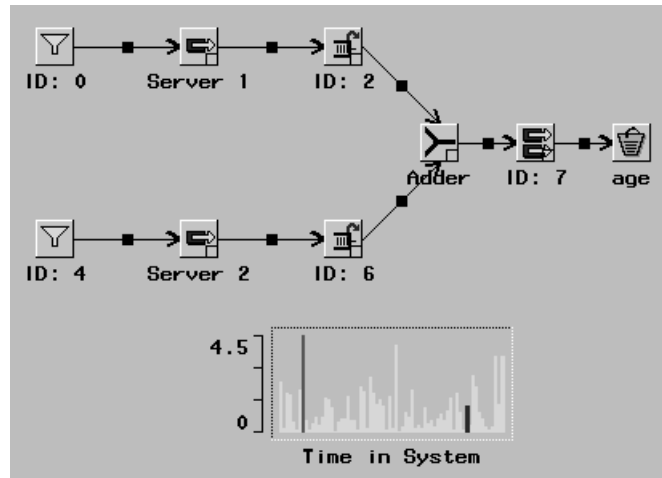


Figure 9.19. Assembly Unit

Figure 9.19 shows the assembly of two subunits into a larger unit. Each subunit line produces components as modeled by servers 1 and 2. These subunits queue in the buffers at the end of the subunit assembly lines. When the multiple server is free, it requests a transaction from the Adder. The Adder requests one transaction from each of the lines going into it. If there is a transaction available from each of these lines, then it requests one. When all the transactions have arrived at the Adder, it generates a new transaction, which is sent down the arc to the multiple server.

Servers as Resources I

There are instances in which where the system needs to schedule concurrent service from multiple servers on a single transaction. In these situations, you can think of servers as resources that are being utilized by the transactions. For example, in an auto repair facility, several mechanics (modeled as servers) can work on a single car (the transaction) at a time. The Splitter is useful for treating servers as resources and capturing concurrent use of the resources.

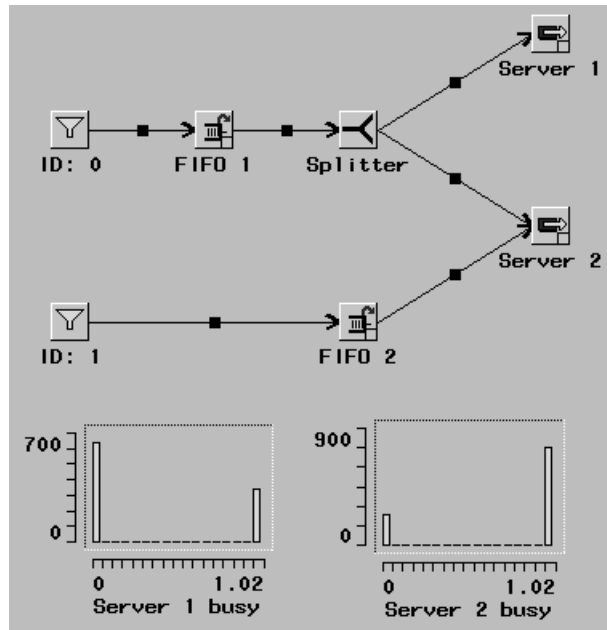


Figure 9.20. Servers as Resources

Figure 9.20 shows a simple model with arrivals from two sources, each sending the transactions into a queue. If the two servers are free and there is a transaction in FIFO 1, then the first transaction inserted into the queue will flow to both the servers and service will start in each. The service times in each of these is independent (unless you construct and use a service time distribution that destroys this independence). When Server 2 becomes free, it requests a transaction. If Server 1 is busy, then the request can only be honored by a transaction in the FIFO 2 queue. When Server 1 becomes free, it requests a transaction that can only be honored if Server 2 is free and there is a transaction in FIFO 1.

Servers as Resources II

In the preceding example, the resources (servers) performed service independently on a transaction. However, there are situations where the resource may be used in a more controlled way. Suppose there are two parallel lines that each require the use of a shared resource (a crane, for example).

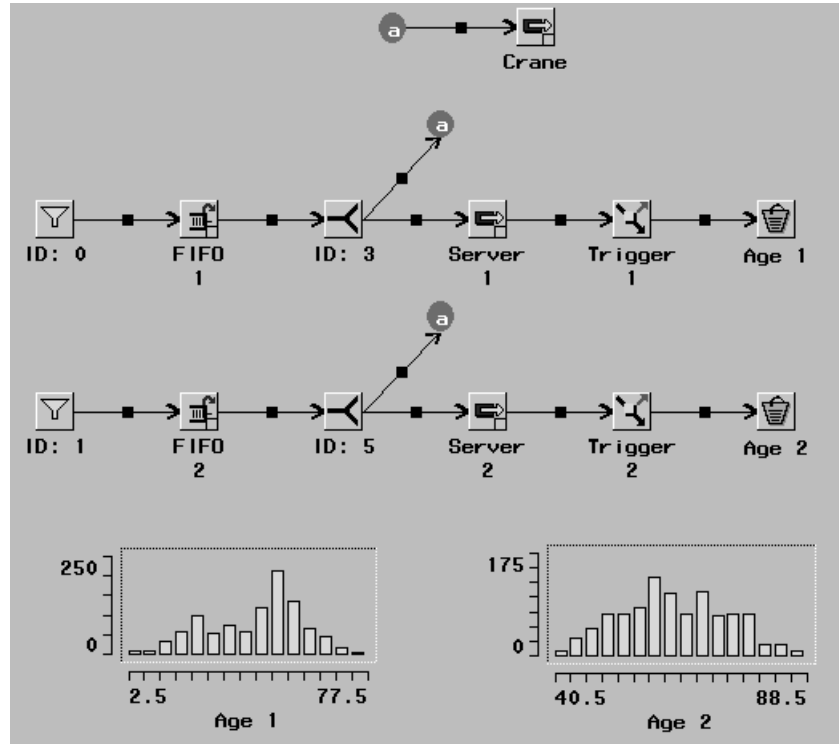


Figure 9.21. Servers as Resources

Figure 9.21 shows such a model. As in the last example, the Splitter is used to capture the shared use of the resources by a transaction. In addition, there is a Trigger after each of the servers in the parallel lines. These triggers release the transaction from service in any other servers. So the time the transaction uses the Crane is the minimum of the time scheduled for Crane use and the line service time. In particular, if the service time in Server 1 is X and the service time specified for the Crane is Y , then the service time that the transaction actually receives in the Crane is $\min(X, Y)$. This occurs because either the transaction finishes with the Crane before it is done with service in Server 1 (or Server 2) or it finishes with service in Server 1 (or Server 2) before the Crane service is completed. In this case, Trigger 1 (or Trigger 2) sends the “RemoveFromServers” messages (see Figure 3.2), which removes that transaction from any servers in which it may be receiving service. In this case, the transaction can be explicitly removed from service by the Crane.

Special Routing

There are other ways that independent streams can share resources. One is illustrated in Figure 9.22.

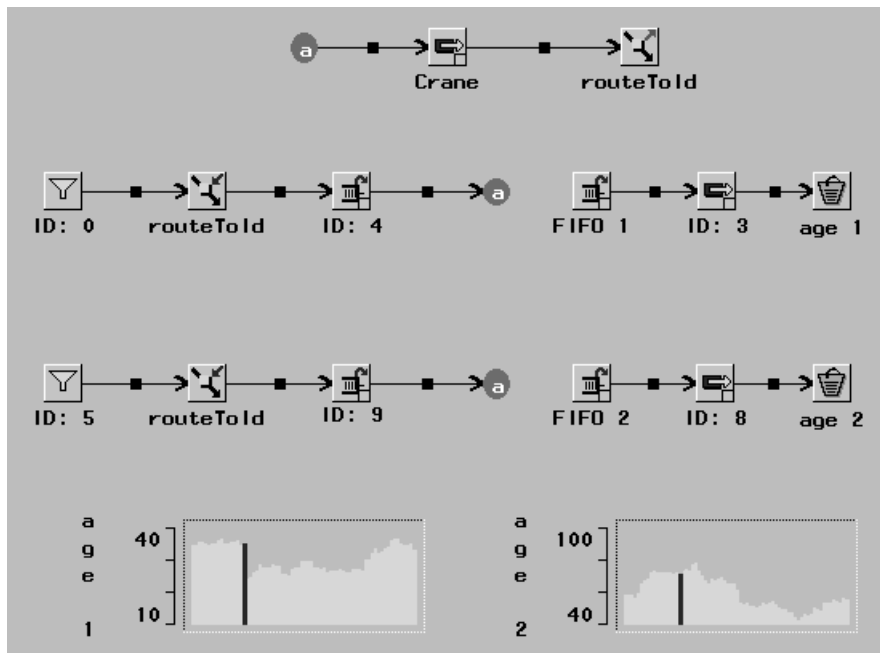


Figure 9.22. Special Routing

Here the Modifiers labeled “routeToId” set the attribute “routeToId” to the id of “FIFO 1” or “FIFO 2.” When the transaction finishes with the service of the “Crane” and traverses to the Trigger labeled “routeToId,” it is routed to the component whose id is in its attribute “routeToId.” This is another way that transactions can be routed through the network.

The correct bibliographic citation for this manual is as follows: SAS Institute Inc., *SAS/OR® User's Guide: QSIM Application, Version 8*, Cary, NC: SAS Institute Inc., 1999. 110 pp.

SAS/OR® User's Guide: QSIM Application, Version 8

Copyright © 1999 SAS Institute Inc., Cary, NC, USA.

ISBN 1-58025-488-8

All rights reserved. Printed in the United States of America. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, by any form or by any means, electronic, mechanical, photocopying, or otherwise, without the prior written permission of the publisher, SAS Institute, Inc.

U.S. Government Restricted Rights Notice. Use, duplication, or disclosure of the software by the government is subject to restrictions as set forth in FAR 52.227-19 Commercial Computer Software-Restricted Rights (June 1987).

SAS Institute Inc., SAS Campus Drive, Cary, North Carolina 27513.

1st printing, October 1999

SAS® and all other SAS Institute Inc. product or service names are registered trademarks or trademarks of SAS Institute Inc. in the USA and other countries.® indicates USA registration.

IBM®, ACF/VTAM®, AIX®, APPN®, MVS/ESA®, OS/2®, OS/390®, VM/ESA®, and VTAM® are registered trademarks or trademarks of International Business Machines Corporation.® indicates USA registration.

Other brand and product names are registered trademarks or trademarks of their respective companies.

The Institute is a private company devoted to the support and further development of its software and related services.