Perfect Sampling for Queues and Network Models

DUNCAN J. MURDOCH

University of Western Ontario

and

GLEN TAKAHARA

Queen's University

We review Propp and Wilson's [1996] CFTP algorithm and Wilson's [2000] ROCFTP algorithm. We then use these to construct perfect samplers for several queueing and network models: Poisson arrivals and exponential service times, several types of customers, and a trunk reservation protocol for accepting new customers; a similar protocol on a network switching model; a queue with a general arrival process; and a queue with both general arrivals and service times. Our samplers give effective ways to generate random samples from the steady-state distributions of these queues.

Categories and Subject Descriptors: I.6.8 [**Simulation and Modelling**]: Types of Simulation—*Monte Carlo*; I.6.3 [**Simulation and Modelling**]: Applications

---

Author's addresses: D. J. Murdoch, Dept. of Stat. and Act. Sci., University of Western Ontario, London, Ontario, Canada N6A 2C8; G. Takahara, Dept. of Math and Stats, Queen's University, Kingston, Ontario, Canada K7L 3N6.

General Terms: Algorithms

Additional Key Words and Phrases: Coupling from the past, perfect simulation, queues, networks

1. INTRODUCTION

The aim of *perfect sampling* is to take approximate simulation methods (e.g. simple Markov chain Monte Carlo) and to simulate from the limiting distribution, usually using some sort of coupling method. In this paper we demonstrate how to apply these methods to queues and network models.

There is a huge literature on simulation of queues and network models in general. In particular, Andradóttir and Ott [1995] and Andradóttir and Hosseini-Nasab [2003] described methods of simulation of Markovian models on parallel processors that involved coupling and reduction of initialization bias. Their work did not use perfect sampling as in Propp and Wilson [1996] (described below), and did not yield samples drawn perfectly from the steady-state model, but the coupling they used is the same as the coupling used in Section 2.1 below.

1.1 Algorithms

The *coupling from the past* (CFTP) algorithm was introduced by Propp and Wilson [1996]. It allows a simulation of a Markov chain to be used to produce a simulated value whose distribution is the limiting distribution of the chain. Since Markov chain Monte Carlo algorithms are often used to produce approximate inference about the steady state distribution, CFTP and related algorithms have come to be known as *exact* or *perfect* simulation.

The simplest description of CFTP applies to an aperiodic discrete time Markov chain, though it can be applied much more generally; indeed, our applications below are more general. We suppose that the Markov chain X_t may be simulated using a recursive formulation

$$X_{t+1} = \phi(X_t, U_{t+1}) \tag{1}$$

where U_{t+1} are i.i.d. from some known distribution. The algorithm proceeds as follows:

(1) Conceptually, generate the sequence $U_0, U_{-1}, \ldots$. In actual practice these values are generated as needed.

(2) Search for a time $-T < 0$ such that paths started at every possible starting point at that time, updated using (1) repeatedly, all produce the same simulated value for X_0 (i.e. they *coalesce*).

(3) Output the common value X_0.

Typically the search for $-T$ proceeds by trying some value $-T_0$, then trying $-2T_0$ if that fails, and continually doubling until success is achieved. The art of creating CFTP algorithms lies in constructing $\phi(\cdot, \cdot)$ so that the search terminates. It is easy to see that if the probability of termination is greater than zero for some finite $-T$, then the search terminates with probability 1: each new segment further back in time is independent of the already-examined tail segment. If the state space is very large (e.g. a continuum), then tricks such as monotonicity [Propp and Wilson 1996; see discussion below] and gamma couplers [Murdoch and Green 1998] are used to reduce the amount of computation to manageable levels.

For continuous time jump processes, we need a more complicated formalism,

but the same general ideas apply. We need an easily simulated point process U_t (e.g., a constant rate Poisson process) to drive X_t. For CFTP, we need to be able to simulate U_t in a time interval $[-T, 0]$, and then, conditional on these values, simulate it in an earlier interval $[-2T, -T]$ (and so on recursively). The value of X_t at time 0 must be completely determined by the value of X_{-T} and the realization of U_t on $[-T, 0]$. The general CFTP algorithm then simulates U_t on $[-T, 0]$ and doubles T until every possible value of X_{-T} is updated to the same output X_0 (coalescence). In this general setting CFTP will not necessarily terminate. For the cases where CFTP terminates with probability 1, the value of X_0 is determined by the realization of the U_t process on some finite interval $[-T, 0]$; the same value would be drawn when started at any value at any earlier time. Thus its distribution is the same as the limiting distribution of the chain.

Proving that CFTP terminates is typically straightforward in our settings. If realizations of U_t are independent in non-overlapping intervals (as with a Poisson process), then CFTP will terminate provided the probability of coalescence is positive. For more general conditions and a rigorous proof of coupling in finite time see Theorem 8.2 of Chapter 10 in Thorisson [2000]. We consider a more general case in Section 2.4 below.

At this point, most readers newly exposed to CFTP will be wondering about the complication of going backwards in time. Why not just run coupled chains forwards until coalescence as in Andradóttir and Ott [1995]? The answer is that the state at the time of coalescence is typically not a draw from the steady-state of the chain. CFTP gives the same output no matter how large T is chosen to be;

thus the output must be a draw from the limiting distribution. Forward coupled chains converge to the limiting distribution, but the distribution at any fixed time point, or fixed time after a coalescence, will generally not be in steady-state: the event of coalescence may bias the distribution.

Nevertheless, we would like to avoid one of the difficulties with CFTP: it requires the U_t values to be used repeatedly. When an attempt for coalescence fails and T is increased, it is important that the U_t values already generated (i.e. those that fall in $[-T, 0]$) are re-used, as this guarantees that the output value is the same regardless of the search strategy. In practice, this is usually done in one of two ways. Either the U_t values can be stored as generated, or they can be regenerated as needed, by storing random number seeds and re-using them. Both require an unbounded amount of storage availability, with the needs of the former growing exponentially faster than those of the latter. The latter uses pseudo-random number generators (PRNGs) in ways for which they were not likely designed or tested: whereas many PRNGs are extensively tested and found to be good approximations to i.i.d. samples in the order generated, it is not always clear that they will be as good when the values are re-ordered. If hardware based random number generators are employed, the storage strategy is the only one available.

Read-once CFTP (ROCFTP) addresses these concerns [Wilson 2000]. This variation on the algorithm proceeds as follows. Again the description is for aperiodic discrete time Markov chains.

(1) Choose a block size T.

(2) Simulate blocks of updates of T steps forward in time until a block is found in

which coalescence occurs, say block n, consisting of the T steps between the $T+1$ states at times $(n-1)T$ through nT. Set X_{nT} (the "special path") to the value at the end of this block.

(3) Continue to simulate blocks forward in time, following paths from all states at the start of each block, but paying particular attention to the special path, until another coalescent block is detected, say block $m > n$.

(4) Output $X_{(m-1)T}$, the value of the special path at the *start* of the next coalescent block.

This algorithm requires that the U_t values within non-overlapping time blocks are mutually independent, so the events that block i coalesces, which we denote by C_i, $i = 1, \ldots$, are i.i.d. events. ROCFTP can then be seen as equivalent to CFTP with a relabelling of the time axis, where time $(m-1)T$ is relabelled as time 0. The output value depends only on U_t values in blocks $n, \ldots, m-1$, which are independent of the values in blocks m or greater; by the time-invariance of the structure, the relabelling of the time axis has no effect on the distribution of the output value. This is a discrete-time version of the PASTA property [Wolff 1982]. The distribution of a stationary process at the fixed time 0 is the same as its distribution just before a randomly sampled time, provided the sampling time is independent of the history of the process. The reason ROCFTP requires two coalescence events instead of the one required by CFTP is that the C_i events play two roles: the second one determines the sampling time, and the first guarantees that the value of the special path will be known then.

Because the blocks are independent, ROCFTP will always terminate provided

$P(C_i)$ is positive. Wilson [2000] shows that ROCFTP is most efficient when T is chosen near the median forward coalescence time so that $P(C_i)$ is around 1/2; in that case it is typically about 1/3 less time-efficient than CFTP, though it can be more efficient in certain special cases. As discussed above, it generally makes more efficient use of memory.

As we will see below, implementations of CFTP in which the U_t values are not independent between blocks cannot easily be redone as ROCFTP. However, in cases where independent blocks are available, we would generally recommend ROCFTP over CFTP. It is simpler and/or less resource-intensive to program.

1.2 What Distribution to Simulate?

When we say we are producing a draw from the steady-state distribution of a queue or network model, we need to ask: which steady-state distribution? As is well-known, the time-average steady-state (i.e. the limit of the distribution of the state at time t, as $t \to \infty$) may be different from the distribution of the state at the n^{th} arrival, as $n \to \infty$. The descriptions of CFTP and ROCFTP apply to time-average steady-states, as time 0 is a fixed time in a long running (from the past) Markov chain. In the examples in Sections 2.1, 2.2 and 2.3 below there is no distinction, because of the Poisson arrival assumption. In Section 2.4 we show how to simulate both steady states.

2. EXAMPLES

2.1 Example: M/M/1/C Queue

Consider a queue with a single server and a finite buffer, with total capacity $C+1$. Customers who arrive when the buffer is full are lost. Arrivals occur in a Poisson process at rate λ; service is at rate μ (i.e. there are exponential service times with mean $1/\mu$.)

The state of this queue is entirely represented by X, the number of customers currently in the system. Transitions to a higher state occur at rate λ, provided $X < C+1$; transitions to a lower state occur at rate μ provided $X > 0$. Steady-state analysis of this queue is straightforward; for example, see Gross and Harris [1998 §2.4]. However, we will construct a perfect sampler for it as a "warm-up".

Both CFTP or ROCFTP are straightforward to apply to this queue. There is only a finite set of states; one could define random updates for each possible state, and follow them all forward through time until they coalesced. However, if C is large, this will be impractical; there will be too many states to follow, and it may take too long for them to coalesce.

One approach often used to reduce the amount of computation is to apply a (partial) order to the state space, and to choose an update function ϕ that preserves this order, i.e. $x \preceq y$ implies $\phi(x,u) \preceq \phi(y,u)$ for all u, where "$\preceq$" represents the partial order relation. This property is known as *monotonicity* of the update function. Under monotonicity only maximal and minimal elements need to be followed; all others are sandwiched between them. The monotonicity approach works well here if we apply the same potential updates to every state, because the

natural integer ordering "$\leq$" on states is preserved under these updates. Thus we can determine coalescence by following states $X = 0$ and $X = C + 1$ until those two states coalesce.

One minor complication is that the U_t process (here the arrival and service events) must be generated independently of the current state of the system, but when the queue is in state 0 no service events occur. However, this is dealt with simply by generating service events uniformly across time, even when the queue is empty, and having the queue reject them if it is in state 0, just as it rejects arrivals when it is in state $C + 1$.

With these choices, we implement ROCFTP with block size T as shown in Fig. 1, where we use $L \leq X \leq R$ to represent the state at time t of the lower limit, steady-state and upper limit chains respectively, $\text{Exp}(\lambda + \mu)$ and $\text{Unif}(0, 1)$ are randomly generated values from the Exponential distribution with rate $\lambda + \mu$ and the uniform distribution on $[0, 1]$ respectively. The arrival and service events are generated by creating unlabelled events at the sum of the respective rates, and then randomly labelling them as arrival or service events. The last event generated in *MMblockupdate* will generally fall outside the block; by the memorylessness of Poisson arrivals, it is safely discarded, and a new event generated the next time this subroutine is called.

An implementation of CFTP would also be straightforward. There the main implementation issue is how the events are generated and saved. A simple pseudocode implementation that assumes they are stored and re-used is given in Murdoch and Green [1998].

```
MMrocftp(T):
    repeat                          Find first coalescence
        L ← 0                       Initialize
        X ← undefined
        R ← C + 1
        MMblockupdate(T,L,X,R)
    until L = R
    X ← R
    repeat                          Find next coalescence
        L ← 0
        R ← C + 1
        Y ← X                       Save start value
        MMblockupdate(T,L,X,R)
    until L = R
    return Y
```

Fig. 1. ROCFTP algorithm for M/M/1/C Queue. (Continued on next page)

2.2 Example: M/M/C/C Queue with Trunk Reservations

In this section we generalize the M/M/1/C queue of Section 2.1 to a priority queue with capacity C, and p customer types. Customer type $i = 1, \ldots, p$ arrives in a Poisson process with rate λ_i and uses b_i units of capacity when accepted for service. However, there is a trunk reservation of r_i units: an arriving customer of type i will only be accepted if there are currently at least $r_i + b_i$ units of spare capacity on the server. Customers who are not accepted are lost. Service times are exponential with service rate μ_i for type i customers. Because of the exponential interarrival

```
MMblockupdate(T, L, X, R):          Block update function
    t ← 0                           time within block
    repeat
        t ← t + Exp(λ + μ)          Generate time of next event
        if t > T then
            return modified (L, X, R)
        else
            U ← Unif(0, 1)          Used to determine event type
            L ← MMupdate(L, U)
            X ← MMupdate(X, U)
            R ← MMupdate(R, U)
    until return
```

```
MMupdate(x,u):                      State update function
    if u < λ/(λ + μ) then
        x ← min(x + 1, C + 1)       Arrival event
    else
        x ← max(x − 1, 0)           Service event
    Return x
```

Fig. 1. (Continued from previous page) ROCFTP algorithm for M/M/1/C Queue.

and service times, we can represent the state as the vector $X = (X^{(1)}, \ldots, X^{(p)})$, $X^{(i)} \in Z^+$, $\sum_i X^{(i)} b_i \leq C$, giving the counts of customers of each type currently being serviced.

With all $r_i = 0$, an explicit analysis of this queue is straightforward [Ross 1995, Ch. 2]. However, positive trunk reservations complicate the analysis significantly.

For example, if $p = 2$ and $r_1 < r_2$, then there may be states near full capacity that can be reached by the departure of customers of type 2, but not by type 2 arrivals. Since the embedded Markov chain of transitions is not reversible, computation of the steady-state distribution is difficult, and simulation may be the best approach to find the steady-state properties of the queue.

Here we again have only a finite set of states, but the number of states is large. Unfortunately, a monotone update function does not appear to be available: it is not obvious how to find the required partial order on this state space.

However, partial order ideas can be used to reduce the amount of computation substantially. First, we define U_t to be a marked Poisson process with rates $(\lambda_i, i = 1, \ldots, p; \mu_{ij}, i = 1, \ldots, p, j = 1, \ldots, N_i)$ where $N_i = \lfloor (C - r_i)/b_i \rfloor$ represents the maximum possible count of type i customers. The λ_i rates correspond to arrivals of each respective type of customer; the μ_{ij} rates measure the departures of the j^{th} type i customer in the system, and satisfy $\mu_{ij} = \mu_i$. We call the U_t events the "labelled arrival and service times". To use this process to update a given state X_t, we wait for the next event. If it is an arrival, we accept it if X_t has spare capacity for that customer type. If it is a μ_{ij} departure, we reduce $X_t^{(i)}$ by 1 if $X_t^{(i)} \geq j$, and otherwise do nothing. (This process can also be thought of as generating departure events for the queue with N_i customers and filtering these down to the actual number of customers in the queue.)

With this update rule, we have near monotonicity with respect to the componentwise partial order defined by $X \preceq Y$ if $X^{(i)} \leq Y^{(i)}$ for all i. It is easy to verify that the partial order is preserved across all departure events. It is also preserved across

most arrivals, with a few exceptions. For example, if $X \preceq Y$ with $X^{(i)} = Y^{(i)}$ when an arrival of type i occurs, it may be that X accepts the customer, while Y is filled to capacity and rejects it, breaking the order. However, if $X^{(i)} < Y^{(i)}$ or $X^{(i)}$ and $Y^{(i)}$ both make the same accept/reject decision about the arrival, then the order will be preserved.

Another problem is that while the empty queue $\hat{0} = (0, \ldots, 0)$ is a minimal element with respect to $\preceq$, there is no maximal element.

We deal with the second problem first, by extending the state space to $\mathcal{X} = \{(n_1, \ldots, n_p) \mid \forall i \ 0 \leq n_i \leq N_i\}$, i.e. by adding "illegal" states which exceed the capacity of the queue. This gives us a maximal element $\hat{1} = (N_1, \ldots, N_p)$. We define the response of these states to arrival and departure events in exactly the same way as for the legal states. Since none of the new states will ever accept an arrival, they are all transient and the steady state distribution of the new chain is the same as the steady state distribution of the original chain.

There are several ways to deal with the incomplete monotonicity. One is to create a new process operating on (hyper-)rectangles of states, i.e. with state space $\mathcal{R} = \{(A, B) : A, B \in \mathcal{X}, A \preceq B\}$. We use the same random inputs to update the rectangles according to the following rule, designed to ensure that a point starting out in the rectangle will remain in the rectangle: if $R_t = (A_t, B_t)$ then $R_{t+1} = (\min_{A_t \preceq x \preceq B_t} \phi(x, U_t), \max_{A_t \preceq x \preceq B_t} \phi(x, U_t))$ (Fig. 2). We note two things about this construction. First, if we start the rectangle chain at $(\hat{0}, \hat{1})$ and it eventually evolves to a single point (i.e. a state (A, B) with $A = B$), then we know that the chain on $\mathcal{X}$ started at any point will be in state A at the same time, i.e.

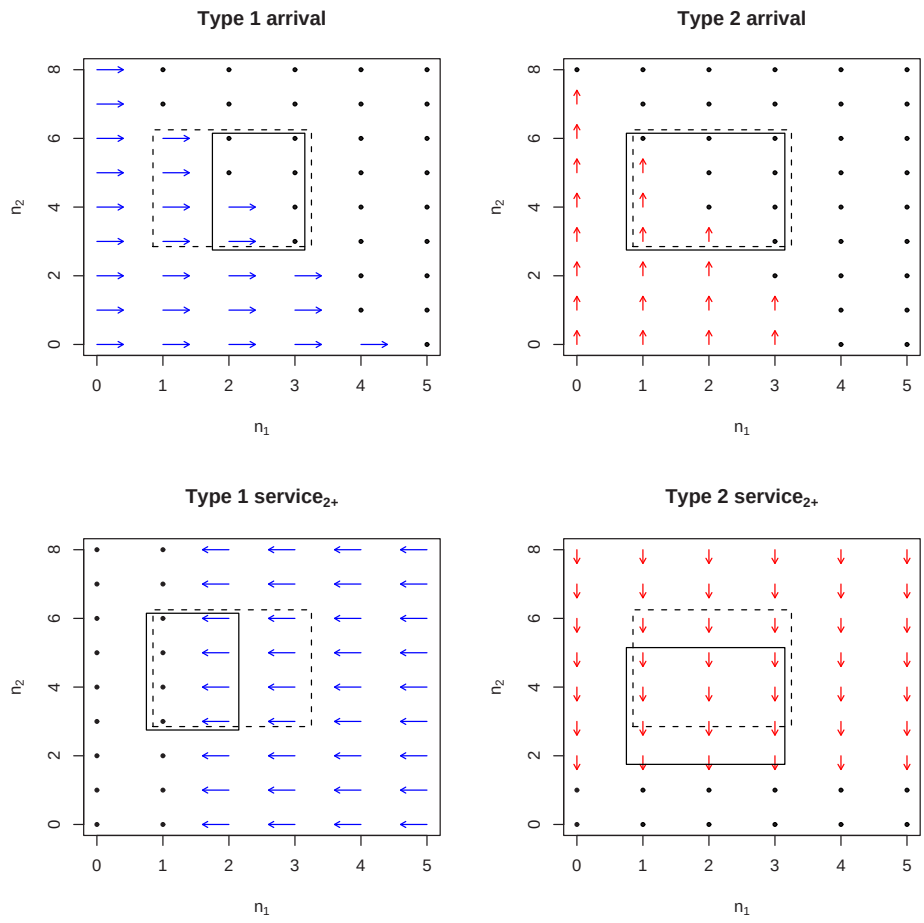

Fig. 2. An illustration of some possible events in the trunk reservation model with $C = 10$, $p = 2$, $b = (2, 1)$ and $r = (0, 2)$. A type i service$_{k+}$ event affects only states with k or more type i customers present. The dots indicate states that would not move; the arrows indicate moves that would take place. The dotted rectangle outlines a typical R_t group of states, and the solid rectangle shows the corresponding outline of R_{t+1}. Both rectangles have been expanded slightly to avoid overplotting on the dots, arrows or each other.

the coalescence required by CFTP has occurred.

Second, updates of rectangles require updates of just two states. Because of the

partial monotonicity, departure events may be applied directly to each of the two extreme states in the rectangle, and all states in the rectangle are automatically bounded by the result. Arrivals of customers of type i should increase $B_t^{(i)}$ if there are any states X in the rectangle with $X^{(i)} = B_t^{(i)}$ and with sufficient free capacity to accept an arrival. We check this by updating the state X constructed to be equal to A_t in all coordinates except coordinate i, where $X^{(i)} = B_t^{(i)}$ is used, since this state has the greatest free capacity of any such state. Similarly, those same arrivals will increase A_t if all states X in the rectangle with $X^{(i)} = A_t^{(i)}$ accept the arrival; this may be tested by constructing X equal to B_t in all coordinates except i, where $X^{(i)} = A_t^{(i)}$ is used, the state with the least free capacity. (In two dimensions, these constructed points are the upper left and lower right points in the rectangle, and the algorithm may be defined to follow just their paths; for larger p, the constructed point depends on i.)

Fig. 3 illustrates a typical run of the CFTP algorithm for this model. In this run, it took 3 attempts to find the starting point ($t = -4$) from which all points coalesced. In the figure we show the bounds on the system load $\sum b_i X_t^{(i)}$ corresponding to the lower left and upper right corners of the rectangles shown in Fig. 2, as they evolve over time. When those two corners have the same load, they correspond to the same state: coalescence has occurred. The indicated point at time $t = 0$ is a draw from the steady-state distribution for this queue. To generate additional independent points from the steady-state distribution, the whole process would be repeated.

Fig. 4 illustrates the ROCFTP algorithm on the same model. Not shown is an

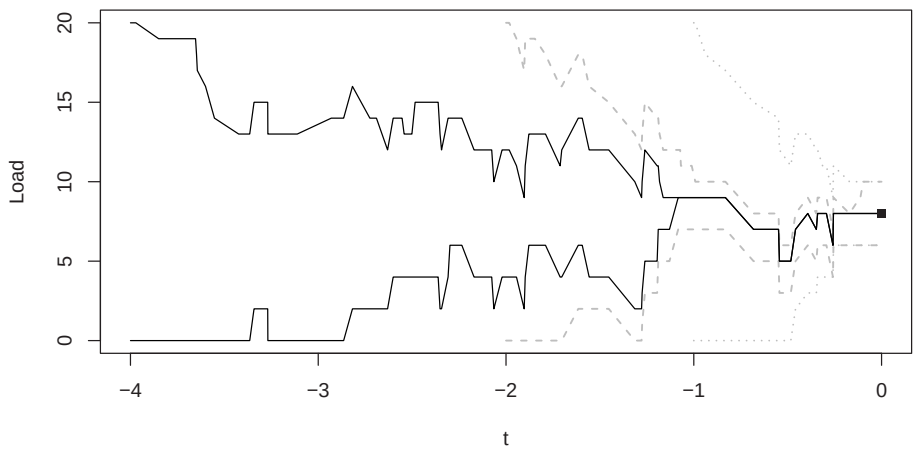

Fig. 3. A sample run of the CFTP algorithm for the M/M/C/C queue with trunk reservations using the same parameters as in Fig. 2. The dashed and dotted gray lines show failed attempts to find coalescence; the solid black line is the attempt that succeeded, resulting in the point at $t = 0$ being output.

initial series of training runs from which the block size of $T = 3.7$ was set as the median forward coalescence time of 9 runs. The figure starts with a block which failed to coalesce; this is followed by a successful coalescence, from which the special path is started. The third block failed and the fourth succeeded in coalescence; thus the value on the special path at the start of the fourth block would be the output. To generate additional independent draws, the special path could be started at the value shown at the right side of the figure; additional training or initialization would not be necessary. Fig. 5 shows a sample of 1000 i.i.d. draws.

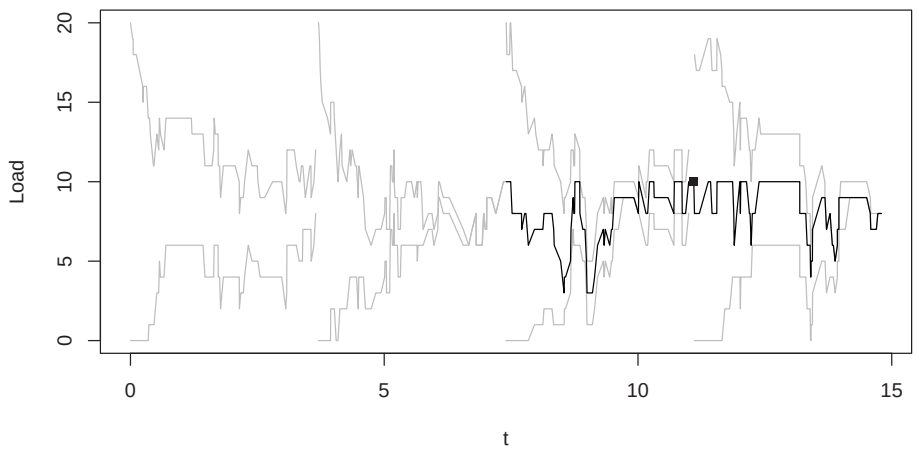

Fig. 4. A sample run of the ROCFTP algorithm for the M/M/C/C queue with trunk reservations using the same parameters as in Fig. 2. The gray lines show the bounding paths; the solid black line is the special path from which draws are taken.

2.3 Example: Circuit Switching Network with Trunk Reservations

Consider a circuit switching network with capacity limited links (e.g. Fig. 6). The network is an undirected graph with I edges. Messages are sent between the nodes of the graph across J different routes $r_1, \ldots, r_J$. Link i has capacity C_i units of bandwidth; each connection on route r_j holds A_{ji} units of bandwidth on link i. Again we assume a trunk reservation system: if there are not $A_{ji} + R_{ji}$, $i \in r_j$, units of unused bandwidth on link i when an incoming route r_j connection request arrives, the connection is blocked and lost to the system. We assume Poisson arrivals at rate λ_j and exponential service times with rate μ_j for connections on route r_j.

This model is very similar to the queue considered above. The state space may

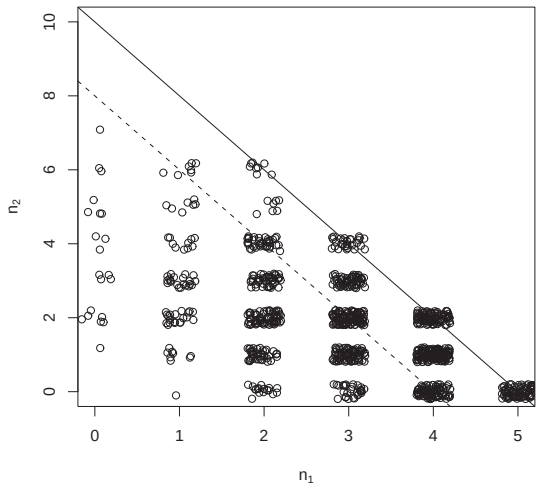

Fig. 5. A sample of 1000 i.i.d. draws from the M/M/C/C queue with trunk reservations using the same parameters as in Fig. 2. The solid line shows the upper bound on the state space; the dashed line shows where the trunk reservations come into play. The points are the ROCFTP output values jittered to avoid overlap.

be represented as the vector $(X^{(1)}, \ldots, X^{(J)})$ of counts $X^{(j)}$ of customers using route r_j. Instead of a simplex of legal states, the capacity limits lead to allowed states consisting of a convex polytope intersected with the integer lattice; again, we extend this to a rectangular region with upper limits $N_j = \min_i \lfloor C_i / A_{ji} \rfloor$.

2.4 Example: G/M/C/C Queue with Trunk Reservations

The previous implementations appeared to depend on exponential interarrival and service times to simplify the state space. However, a general interarrival distribution can be handled without much more difficulty.

We return to the example from Section 2.2, but this time assume that the in-

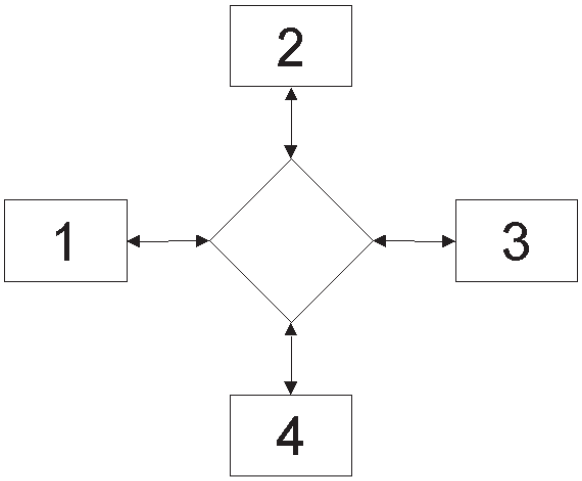

Fig. 6. A simple network model with 5 nodes (4 communicating nodes and a central switch) and 4 edges (capacity limited trunk lines).

terarrival times for customers of type i are i.i.d. with distribution function $F_i(\cdot)$, which we assume has a finite variance and is non-arithmetic [Feller 1966, p.355]. The arrival processes of different types of customers are assumed to be independent.

To be specific, we will assume Pareto$(a_i,\ c_i)$ interarrival times, with $F_i(x) = 1 - (a_i/(a_i + x))^{c_i}$ for $x > 0$. The parameter $c_i > 1$ is chosen to control the "burstiness" of the arrival process, with lower values more bursty; our simulations use $c_i = 1.1$ for all i. The parameter a_i is chosen to match the mean interarrival time $a_i/(c_i - 1)$ to the corresponding value $1/\lambda_i$ in the Section 2.2 simulation. We will label the average arrival rate as λ_i.

At steady-state, the time since the last type i arrival will follow the so-called "current life" distribution, with density function $[1 - F_i(x)]\lambda_i$. For our Pareto arrivals, this distribution is Pareto$(a_i,\ c_i - 1)$.

We still assume exponential service times. Then in order that we have a Markov

process, it is sufficient to supplement the counts $X^{(i)}$ of each type of customer currently in the system with the times since the last arrival of each type of customer.

Since arrivals are independent of the service process, we can easily simulate the arrival process at steady-state, and use this to drive the CFTP algorithm.

(1) Simulate Y_i, $i = 1, \ldots, p$ from the density $(1 - F_i(x))\lambda_i$. Set the most recent arrival time of customers of type i to $A_1^{(i)} = -Y_i$ for $i = 1, \ldots, p$.

(2) Choose $-T < 0$.

(3) Simulate arrival times $A_j^{(i)}$, $j = 2, \ldots$ of each customer type $i = 1, \ldots, p$ backwards until $A_{j_i}^{(i)} < -T$ for each i. These are simulated recursively by drawing the interarrival times from F_i. Save these arrival times, and the current value of the random number seed.

(4) Generate a labelled service time following time $-T$ as in Section 2.2.

(5) Set the current hyper-rectangle of states to cover all possible states.

(6) Repeat the following loop forward until time 0 is reached.

   (a) Choose whichever of the p arrival times or the labelled service time comes first. With Pareto interarrival times and exponential service times ties will not occur, but in general ties could arise. In that case ties should be broken as they would be in the system being modelled, e.g. higher priority customers could be treated as arriving first.

   (b) Process that event as in Section 2.2 to update the hyper-rectangle.

   (c) Replace an arrival time by choosing the next saved $A_{j_i}^{(i)}$ value. If none remain, use an arbitrary positive value instead. Replace a service time by generating a new labelled service time using the appropriate exponential

inter-service time distribution.

(7) Check for coalescence of the hyper-rectangle to a single state. If it has occurred, output the state at time 0. Otherwise, double T, and repeat from step 3.

If we are interested in simulating the state at the time of a type i arrival, we simply replace the distribution used in step 1 for that type with F_i (i.e. we assume that there was a type i arrival just after time 0).

This algorithm requires that large amounts of storage be devoted to the $A_j^{(i)}$ values. An alternative implementation with fixed storage would regenerate them as needed, by restoring the random number generator seed and cycling through the values. However, this could be very time-consuming.

Unfortunately, implementing ROCFTP is not straightforward here. The arrivals within blocks of time of a fixed length are not independent of other blocks. If there were only one customer type, the blocks could be defined in terms of the number of arrivals of that type of customer; then independence would be achieved. However, this is not possible with $p > 1$.

It might be possible to implement ROCFTP using the gamma coupling techniques of Murdoch and Green [1998]. These would require expanding the state space to include the times since the last arrival at the start of each block, and coupling all possible conditional updates. However, these are difficult to implement, and not always successful, in that an infinite number of paths might result. It seems simpler to use the CFTP algorithm in this case. Termination is guaranteed because there is a non-zero probability of coalescence in any interval (e.g. realized service times fast enough to empty even a full queue after the last arrival).

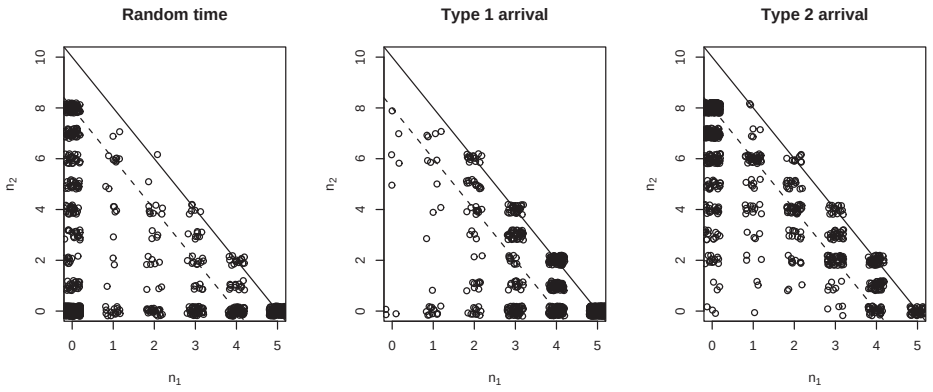

Fig. 7. Simulations of the G/M/C/C queue with trunk reservations. The plots show the three different steady-state distributions under bursty Pareto arrivals.

Fig. 7 shows the output from 3 runs of the CFTP algorithm for this queue, illustrating both qualitative and quantitative differences between this queue and the M/M/C/C queue illustrated in Fig. 5. On the left, the time-average steady-state distribution has been sampled. The queue spends about forty percent of the time completely empty, as shown by the large concentration of points at $(0, 0)$. Fifty percent of the time there is only one type of customer in the queue. The centre and right plots show that arriving customers see very different behaviour. Arrivals almost never (only 0.3% of the simulations) see an empty system, and thirty percent of the time see customers of both types. Refusal rates can also be calculated from the simulations by looking at the proportion of simulations on the solid line for type 1 (i.e. 66%), and on or above the dashed line for type 2 (i.e. 72%).

2.5 Example: G/G/C/C Queue

When both arrivals and service times come from general distributions, perfect sampling is more difficult. These chains are Markov only if the arrival times of each individual in the system are included as part of the state space; that makes the state space very large, and in general it appears that gamma coupling techniques [Murdoch and Green 1998] may be necessary. However, in the special case where the service times are bounded, an approach similar to the one above may be successful.

One simple perfect sampler for the case of bounded service times would be to use CFTP as follows. Simulate arrival times backwards from time 0 until a gap in arrivals is observed that is longer than the upper bound on service times. By the end of that gap, the system will be known to be empty. Start a path at that known state and simulate forward in time to time zero. The difficulty with this approach is that such regeneration points may not occur frequently enough; the search would have to go too far into the past to be practical. Here we describe a CFTP scheme which detects coalescence without requiring regeneration.

For simplicity, we assume in this section that there is only one type of customer (i.e. $p = 1$). The methods may be extended to more types. We start by assuming there are C servers and no buffer. Customers arrive in a process with interarrival distribution $F(\cdot)$; those who arrive when a server is free occupy it for a service time with distribution function $G(\cdot)$, while those who arrive when all servers are busy are lost. We make the same assumptions on F and G as in the previous section (i.e., finite variance and non-arithmetic). For the general discussion we will assume without loss of generality that the service time distribution is bounded above by 1;

for concreteness we will assume $G(\cdot)$ is Uniform$(0, 1)$.

The boundedness of the service time distribution is useful because of the following Markov-like property: the state at time t is determined entirely by events (arrivals, their acceptance or rejection, and their service) in the time interval going back to time $t - 1$. To implement CFTP, we can use the same technique as in the previous section to simulate the arrival times backwards from time 0. Whether a particular arrival is accepted or not depends on the number of customers in the system at the time of the arrival, which we do not know: some unknown number of customers (the "old customers") will be left from arrival times which have not been simulated. Furthermore, service completion times for the old customers are not known, other than the fact that they will occur sometime within the first time unit in our simulation.

To handle this, we store the simulation in two parts. First, we store the values of all arrivals in an "arrivals list" $0 > A_1 > A_2 > \cdots$. As per usual in CFTP, we only simulate these values as needed. For each simulated arrival A_j, we store a virtual "service completion time" $s_j > A_j$, i.e. the time at which that individual would be serviced if they were accepted into the system at time A_j. When doing CFTP, we generate enough A_j's to know all arrival events on the interval $[-T, 0]$.

The second part of the simulation is the storage of the possible current states of the system at an instant $t \in [-T, 0]$. Possible states are of the form $X_t = (n, \{j_i\}_{i=1}^m)$, where $\{0, \ldots, n\}$ is the range of possible numbers of old customers still in the system, m is the number of new customers currently in the system, and $j_1, \ldots, j_m$ are the indices of the customers' arrival times from the the A_j array. A

state of the queue changes at the unknown times of service for old customers (when n decreases by one, reaching 0 by $-T+1$), at the times of accepted arrivals (when m increases by one, and the new service time is added to the service completion list), and at the times s_{j_i} in the list, when m decreases by one and the corresponding j_i is dropped. However, because the service times for old customers are unobserved, we cannot update n until $t = -T + 1$, at which point it becomes 0.

The algorithm is then described informally as follows:

(1) Simulate Y from the density $(1 - F(x))\lambda$. Set the most recent arrival time to $A_1 = -Y$.

(2) Choose $-T < 0$.

(3) Simulate arrival times A_j, $j = 2, \ldots$ backwards until $A_j < -T$. These are simulated recursively by drawing the interarrival times from F. Save these arrival times, and the current value of the random number seed.

(4) Simulate the corresponding service durations from G; add these to the A_j values to give s_j.

(5) Set $t = -T$.

(6) Set n to C, set m to 0, and initialize the list of j_i to be empty.

(7) Repeat the following loop forward until time 0 is reached.

   (a) Set the new t to whichever of the arrival times A_j or service times s_{j_i} comes first.

   (b) If $t > -T + 1$, set n to $\{0\}$.

   (c) If we have a service event first and j_i is in the list of active arrivals, delete j_i, and decrement m by 1.

(d) If the arrival comes first, then our action depends on n.

i. If $n + m \geq C$, then it is possible that the arrival will be rejected, so keep the current state as a possible state of the process. Otherwise, delete it.

ii. If $m < C$, then create a state which accepts the arrival (i.e. increments m, adds j to the service time list, and, if necessary, reduces the range of n so that $n + m \leq C$ is guaranteed. If not, do nothing.

Note that either one or two states may be output from the tests above.

(e) Remove any duplicate states, or other redundant states. For example, if two states have the same set of s_j values but different values of n, then it is only necessary to keep the larger value.

(8) Check for coalescence at time 0 by simply counting how many unique states are being followed. If only one, output it; if not double T, and repeat from step 3.

3. CONCLUDING REMARKS

In the models that we considered in this paper, monotonicity was not always achievable, nor were all of the models Markov. Nevertheless, CFTP or ROCFTP allowed relatively straightforward simulation from the steady-state distribution of the chain. What was needed was a simulation that was driven by random inputs which could be simulated backward or forward in time.

Once we had such a simulation we used it to calculate steady-state properties. For example, the simulations illustrated in Figs. 5 and 7 confirm the expected qualitative differences between Poisson arrivals and bursty arrivals, but perfect simulation also allows us to obtain quantitative measures. In practical applications,

this information could be used for efficient planning of telephone networks or other queues.

We have provided a set of examples that give a flavor of how perfect sampling may be applied to queues. A further useful technique may be state space aggregation (called multistage backward coupling in Meng [2000]). In conventional simulation scenarios, one is often interested in relatively few performance statistics, such as blocking probabilities. The probability of being in a given subset of the state space could be efficiently simulated by mapping the state vector to a Bernoulli random variable and declaring "coalescence" based on the values of the Bernoulli sequences being followed. In carrying this out, it is essential that the law of the original process be used to carry out the updates; efficiency comes from the fact that coalescence of the original process to a point is not needed, only coalescence of the Bernoulli sequence. This technique should be efficient even for high dimensional state spaces. A benefit of perfect sampling is that one knows the exact distribution of the target probability estimators.

One might ask whether the extra effort to produce a perfect sample is worthwhile. We believe that the answer is in the affirmative. First, stochastic simulation is an effective way to calculate properties of models that are algebraically intractable, and even for those models where exact algebraic solutions exist, is often more easily implemented and checked than the exact solution. Perfect sampling improves on other stochastic simulation methods by obviating questions of convergence: the "burn-in" stage is not required, since results are guaranteed to be distributed according to the steady-state distribution. It is true that each random value generated

takes a lot of computation time, but Murdoch and Rosenthal [1999] describe ways to combine perfect simulation methods with standard stochastic simulation in order to achieve the unbiasedness of the former and the efficiency of the latter. We believe algorithms in this class should always be employed when they are available.

Acknowledgments

This work was supported in part by NSERC grants to both authors. The authors are grateful to the anonymous referees and editors for some very helpful comments.

REFERENCES

Andradóttir, S. and Hosseini-Nasab, M. 2003. Efficiency of time segmentation parallel simulation of finite Markovian queueing networks. *Operations Research 51*, 272–280.

Andradóttir, S. and Ott, T. J. 1995. Time-segmentation parallel simulation of networks of queues with loss or communication blocking. *ACM Transactions on Modeling and Computer Simulation 5*, 269–305.

Feller, W. 1966. *An Introduction to Probability Theory and its Applications, Vol.2*. John Wiley & Sons, Inc., New York.

Gross, D. and Harris, C. M. 1998. *Fundamentals of Queueing Theory*, 3rd ed. John Wiley & Sons, Inc., New York.

Meng, X.-L. 2000. Towards a more general Propp-Wilson algorithm: Multistage backward coupling. In *Proceedings of the Workshop on MCMC Methods, October 1998*, N. Madras, Ed. Fields Institute, Toronto, 85–93.

Murdoch, D. J. and Green, P. J. 1998. Exact sampling from a continuous state space. *Scandinavian Journal of Statistics 25*, 483–502.

Murdoch, D. J. and Rosenthal, J. S. 1999. Efficient use of exact samples. *Statistics and Computing 10*, 237–243.

Propp, J. G. and Wilson, D. B. 1996. Exact sampling with coupled Markov chains and applications to statistical mechanics. *Random Structures and Algorithms 9*, 223–252.

Ross, K. W. 1995. *Multiservice Loss Models for Broadband Telecommunication Networks*. Springer, London.

Thorisson, H. 2000. *Coupling, Stationarity, and Regeneration*. Springer, Inc., New York.

Wilson, D. B. 2000. How to couple from the past using a read-once source of randomness. *Random Structures and Algorithms 16*, 85–113.

Wolff, R. W. 1982. Poisson arrivals see time averages. *Operations Research 30*, 223–231.

...