Designing a Distributed Queue

Theodore Johnson
Dept. of Computer and Information Science
University of Florida

Abstract

A common paradigm for distributed computing is the producer-consumer model. One set of processes produce objects that are consumed by another set of processes. These objects might be data, resources, or tasks. We present a simple algorithm for implementing a distributed queue. This algorithm has several parameters that need to be tuned, such as the number of probes to find an object, the amount of buffering, and the connectivity between the producers and the consumers. We provide an analytical model that predicts performance, and based on the analytical model we provide recommendations for setting the parameters. Our analytical model is validated by a comparison to simulation results.

Keywords: Distributed Queue, Distributed Data Structure, Scheduling, Performance Analysis, Scientific Computing.

1 Introduction

A common paradigm for distributed computing is the producer-consumer model [2, 1, 5]. The Linda parallel programming language [4] has constructions to allow the easy implementation of shared queues. Other examples include Marionette [13] and Workcrews [16]. In addition, producer-consumer relations are often used in parallel scheduling algorithms [6, 7].

The primary motivation for this work is the UFMulti project [3, 14], a distributed processing system for High Energy Physics. HEP research requires the processing of billions of experimental observations (or *events*) to find the events that contain interesting information [12]. For example, the recent discovery of the sixth quark at Fermilab required the processing of several billion events to find the twelve instances when the sixth quark was definitely observed.

HEP processing is easily parallelizable because each event can be processed independently. The processing typically consists of several stages, with each stage written by a different specialist. For example, PASS-2 processing consists of an event reconstruction stage, in which particle tracks are reconstructed from sensor information, followed by an test to determine if the event is of interest to later processing. So, a typical HEP job consists of a set of processes that read raw events from tape, a set of processes that perform event reconstruction, a set of processes that test the reconstructed events, and a set of processes that write the reconstructed events that pass the test to another tape. Such a scenario is shown in Figure 1. Each event

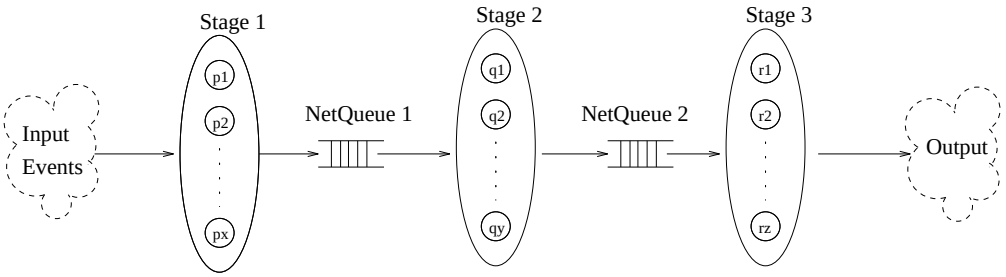

Figure 1: An UFMulti application divided into stages glued together by NetQueues

group is connected by a queue abstraction, which we call Netqueues in the UFMulti system. We are using the research reported in this paper to implement a fully distributed Netqueue.

Some work has been done to implement distributed queues [4, 13, 16, 6, 7, 10]. Manber [11] proposed *concurrent pools*, a shared memory equivalent of a distributed queue. Kotz and Ellis [9] made a performance evaluation of the performance of concurrent pools. The idea of a concurrent pool or a distributed queue is related to techniques for emulating shared memory with a message passing system [15, 8].

In this paper, we present a simple stochastic distributed queue algorithm. This algorithm has features in common with previously proposed algorithms. The contribution of this work is to develop a validated performance model of the stochastic distributed queue algorithm, and to investigate the best parameter settings.

2 The Distributed Queue Algorithm

There are two sets of processes, the *producers* who generate objects, and the *consumers* who use and destroy the objects. A producer repeatedly executes a program that creates a new object, then tries to insert the object into a shared buffer. If there is no room in the shared buffer, the producer blocks until room becomes available. A consumer repeatedly requests an object from the shared buffer, and blocks until an object is delivered. The consumer then processes the object, and repeats its request after finishing the processing. We assume in this paper that the producers and consumers are disjoint, and execute on disjoint processors. This assumption allows us to neglect consideration of the obvious localization optimization (i.e., a consumer always first checks a local producer) when performing the analysis, permitting a cleaner analysis of the algorithm for accessing non-local data. We note that the assumption is likely to be valid in many scenarios (including UFMulti).

An obvious algorithm for implementing the queue is to have a single process store all produced but unconsumed items (i.e., the *centralized buffer* algorithm). This approach has the drawback of requiring a

single process to perform a great deal of work (in receiving and transferring all objects), and to maintain a great deal of storage. So, the centralized buffer solution is not scalable. A more subtle problem is that object descriptions can be quite large, up to several megabytes in UFMulti applications. The centralized buffer solution requires that each object be transferred twice, wasting network bandwidth. A better approach is to distribute the queue, and have each producer store the objects that it produced but which are still unconsumed. With this approach the buffer storage and management is spread among all producers, and objects are transferred only once.

The main complication in developing a distributed queue algorithm is connecting a consumer who requests a new object with a producer who has an object to give. If a consumer has an accurate count of the number of data items available at each producer, the consumer can obtain the object from the producer with an exchange of only two messages (the request and the reply). However, distributing this information requires the exchange of many messages.

One option for distributing the lengths of producer queues is to have each producer multicast to all consumers a description of every enqueue and dequeue event. Such an approach has the advantage of being fully distributed, but requires the exchange of many messages per object. Another approach is to have a single process that acts as a queue manager (i.e., the *queue manager* algorithm). A producer informs the queue manager of every item produced, and a consumer queries the queue manager for a producer with an unallocated object. The queue manager algorithm is better than the centralized buffer algorithm because most of the work in maintaining the distributed queue is distributed among the producers, and objects are transferred only once. However, there is still a large burden placed on a single process. In addition, the algorithm imposes overhead of three control messages (the message from the producer to the queue manager, the request from and reply to the consumer) in addition to the two messages required to perform the transfer (the request to the producer and the reply).

A different approach is to use a fully distributed and stochastically balanced algorithm. Producers maintain their own queues, and have no direct communication with each other. A consumer finds an object by repeatedly probing producers until it finds a producer with an unallocated object, or decides to block at the producer until an object becomes available. We call this algorithm the *stochastic distributed queue* algorithm.

The stochastic distributed queue algorithm has a simple description, but many parameters which can affect performance. The main goal of this paper is to examine the effect of the parameters on the performance of the algorithm. To clarify the discussion, we present the stochastic distributed queue algorithm in pseudo-code.

We start with a description of the parameters:

N	Number of producers.
M	Number of consumers.
buffers$_j$	The number of buffers at producer j.
max_hops	The number of probes a consumer makes before blocking.
p_access[1 .. N]$_i$	p_access[j]$_i$ is the probability that consumer i will choose producer j on a probe.

We use the following conventions to specify the message passing synchronization. When a process sends a message, it executes the following line of code to send a message of type action to destination with parameters parameters:

```
send(destination,action; parameters)
```

When a process waits for an event, it can wait for one of a number of event types to occur. These events can be the reception of a message of a particular type (specified by the action), or an internal event. If more than one event is possible, the code to handle each event is specified along with the parameters passed for the event:

```
wait for event A1, A2, An
        A1 (parameters) :
                code to handle A1
        .....
        An (parameters) :
                code to handle An
```

The protocol at the consumer is to pick a random producer, send a probe to that producer, then wait for the object to be returned (possible from a different producer). The function random uses the distribution specified by its parameter.

```
consumer(self)
    while True
        probe = random(p_access_self)
        hops=1
        send(probe,REQUEST; self,hops)
        wait for a REPLY message from a producer
                REPLY (object) :
        consume(object)
```

We specify the queue management portion of the producer, and we assume that the code that actually produces objects executes in a separate thread. When the production thread creates a new object, it notifies the producer thread and blocks until specifically unblocked by the producer thread. The protocol maintains two data structures: buffer to store produced but unconsumed objects, and blocked-producer to store the identities of blocked consumers. The protocol at a producer is:

```
producer(self)
    initialize the buffer and the blocked-process queue.
    unblock the production thread.
    while TRUE
        wait for a REQUEST message, or for an OBJECT to be produced
            REQUEST (consumer; hops) :
                if the buffer is not empty,
                    obtain an object from the buffer
                    send(consumer,REPLY; object)
                    if the buffer was full,
                        unblock the production thread
                else
                    if hops < max_hops
                        probe = random(p_access_consumer)
                        send(probe,REQUEST; consumer,hops+1)
                    else
                        put consumer in the blocked-process queue
            OBJECT (object) :
                block the production thread
                if the blocked-process queue is not empty,
                    get consumer from blocked-process queue
                    send(consumer,REPLY; object)
                    unblock the production thread
                else
                    put object in buffer
                    if the buffer is not full
                        unblock the production thread
```

The parameters of the stochastic distributed queue represent a wide variety of algorithms, with different resource demands and performance characteristics. The number of producers and consumers, N and M, is defined by the computation to be performed, and is in general not tunable (i.e, is not a parameter available to the algorithm tuner). The most important parameter is max_hops, the maximum number of probes of producers that a consumer will make before blocking. Raising max_hops improves throughput at the cost of additional message passing overhead. More subtle is the effect of buffers_j . Increasing the number of buffers at a producer improves efficiency, but increases the memory overhead of executing the protocol. Finally, p_access_j defines the producers that consumer j will probe, and at what rate. The probability of probing a producer should be proportional to its production rate. In addition, setting $\text{p\_access}_j[i]$ to zero means that consumer j never probes producer i. Therefore, i and j do not need to maintain the overhead to support potential communication. Reliable communication channels (such as BSD sockets) are often scarce resources, and the ability to limit the communications patterns is essential for a scalable implementation. Also, limiting the number of consumers that can probe a producer limits the size of the blocked-process queue.

It is not immediately obvious how to set the parameters of the stochastic distributed queue algorithm.

To permit a logical design, we provide a simple analytical performance model, and execute a performance study.

3 The Simulator

We wrote a simulation to validate the analytical models that we develop. Since the simulator is uniform throughout the study, but is not the focus of the study, we discuss the simulator here.

The simulator accepts as parameters the number of producers, N, the number of consumers M, the sizes of the producer queues $\mathsf{buffers}_j$, and max_hops. In addition, we specify the expected time to send a message, r, the expected time to process a message d, the expected time to produce an object at producer i, $1/\lambda_i$, and the expected time to consume an object at consumer j, $1/\mu_j$. Each of these variables are sampled using an exponential distribution.

For each experiment, we execute the simulation until 1,000,000 objects are consumed. The 95% confidence intervals are within 2%.

4 The Analytical Model

We use the following parameters in the analysis:

- N : Number of producers.
- M : Number of consumers.
- λ_i : The production rate at producer i. More precisely, the time to produce an item has mean $1/\lambda_i$.
- μ_j : The consumption rate at consumer j. More precisely, the time to consume an item has mean $1/\mu_j$.
- h_{max} : the maximum number of hops a request makes before blocking.
- f_i The maximum capacity of the object queue at producer i ($f_i = \mathsf{buffers}_i$).
- r : The average message transit time.
- $p\_access_j[1 \ldots N]$: the probability that a request from consumer j is sent to a given producer.

In addition, we will use the following variables:

- h_{avg} : Average number of hops that a request makes.
- B : Average amount of time that a consumer's request is blocked.

- μ_c : Actual arrival rate of requests to a producer.
- p_{mt} : Probability that the producer has no unallocated objects.
- p_b : Probability that a consumer's request will block if it finds the producer to be empty.
- $p(s)$: Probability that a producer is in state s.
- B_c : Expected time a request is blocked, given that it blocks.

We will develop a series of models, of increasing complexity. The first two models will assume that $p\_access_i = p\_access_j$ for every pair of consumers i and j. The third model will relax this assumption. The first model will assume that every producer produces at the same rate. The subsequent two models will relax this assumption.

4.1 Homogenous Producers

To build the first analytical model, we will assume that all producers have the same production rates. That is, $\lambda_i = \lambda$ for every producer $i = 1 \ldots N$. In addition, we assume that that $p\_access_i = p\_access_j$. Because the consumers make requests in the same proportion to all of the producers, there is no need to distinguish between consumers. Instead, we assume that all consumers have the same consumption rate μ.

The actual rate at which a consumer issues requests is somewhat less than μ, because of the overhead of obtaining objects. In particular, a request must make h_{avg} hops, each of which requires r seconds to be processed. In addition, returning the object requires another hop, costing r seconds. If a request makes h_{max} hops, it will block at the producer for an average of B seconds. The total rate at which consumers issue requests is multiplied by the average number of probes, h_{avg}. If μ_c is the actual rate at which requests arrive at a producer, then we can calculate:

$$\mu_c = \frac{M}{N} \frac{h_{avg}}{1/\mu + h_{avg} * r + B} \tag{1}$$

To make the analysis feasible, we assume that the time to produce an item is exponentially distributed, and that consumer requests arrive in a Poisson process. Since consumer requests are randomly generated from a large population, assuming a Poisson process is not serious. The actual distribution of the production rate might have an impact on the actual performance. We investigate the sensitivity of our model to the inter-production time distribution in the next section, and find that the model remains accurate.

The state of a producer can be modeled as a Markov chain. Since all producers have the same production rate and receive the same rate of requests, we need only model one of the producers, and it will represent

all producers. The state of a producer is represented by an integer s, where $-M \leq s \leq f$. We define the $p(s)$ to be the probability that the producer is in state s. If $s < 0$, then there are $-s$ requests blocked at the producer. If $s > 0$, then the producer has s produced but as yet unallocated objects in its queue. If $s = 0$, there are neither blocked requests or unconsumed items at the producer.

In every state $s < f$ the arrival rate of newly produced items is λ. If $s > 0$, then every arriving request decrements the number of unconsumed objects. Therefore the balance equations for states 0 through f are:

$$\lambda p(s) = \quad \mu_c p(s+1) \quad 0 \leq s < f \tag{2}$$

If $s \leq 0$, then a request will block only if it has made its maximum number of hops. In addition the fact that $s < 0$ tells us that there are s blocked consumers, so the arrival rate of requests is $(M-s)\mu_c/M$. We define p_b to be the probability that a request blocks if it finds the producer queue empty. Then, the balance equations for states $-M$ through 0 are:

$$\lambda p(s) = \quad \tfrac{M+s+1}{M} p_b \mu_c p(s+1) \quad -M \leq s < 0 \tag{3}$$

The state dependent request arrival rates makes the model computationally expensive to solve. We can observe in the usual case, there are only a few blocked consumers at any producer. If M is large, then the state dependent arrival rate buys little in terms of accuracy. We will therefore make the following approximation:

$$\lambda p(s) = \quad p_b \mu_c p(s+1) \quad -M \leq s < 0 \tag{4}$$

This approximation makes the model solution very fast. However, the approximation is not stable if there are many more consumers than producers and p_b is large. We will examine performance in these cases by using simulation. By combining the system of equations 2 and 4 together with the requirement that the state occupancy probabilities sum to 1, we find the solution:

$$p(s) = \begin{cases} \left(\frac{\lambda}{\mu_c}\right)^s p(0) & s > 0 \\ \left(\frac{p_b \mu_c}{\lambda}\right)^{-s} p(0) & s < 0 \\ \frac{1}{\left(\frac{p_b \mu_c}{\lambda}\right)^{M+1} \frac{\lambda}{p_b \mu_c - \lambda} - \frac{p_b \mu_c}{p_b \mu_c - \lambda} - \frac{\mu_c (\lambda/\mu_c)^{f+1}}{\mu_c - \lambda} + \frac{\mu_c}{\mu_c - \lambda}} & s = 0 \end{cases} \tag{5}$$

Note that if $\lambda = \mu_c$, or $\lambda = p_b \mu_c$, then we need to make an exception to handle the degenerate cases. Having found the equations of state in terms of p_b, we must next solve for p_b. A request will block if, after probing $h_{max} - 1$ producers which it found to be empty, it arrives at a producer which is empty. Let p_{mt} be

the probability that a request finds a producer empty. Then:

$$\begin{aligned} p_{mt} = & \quad \sum_{s=-M}^{0} p(s) \\ = & \quad p(0) * \left(1 + \left(\frac{p_b \mu_c}{\lambda}\right)^{M+1} \frac{\lambda}{p_b \mu_c - \lambda} - \frac{p_b \mu_c}{p_b \mu_c - \lambda}\right) \end{aligned} \tag{6}$$

Each consumer request generates h_{avg} requests at the producers. Every time a request arrives at an empty producer, another request is generated, up to h_{max} requests. Therefore,

$$\begin{aligned} h_{avg} = & \quad \sum_{i=1}^{h_{max}} p_{mt}^{i-1} \\ = & \quad (1 - p_{mt}^{h_{max}})/(1 - p_{mt}) \end{aligned} \tag{7}$$

Of the request stream, only those messages that are on probe h_{max} will block. Therefore,

$$p_b = p_{mt}^{h_{max}-1}/h_{avg} \tag{8}$$

We next calculate the average time that a request spends blocked. Suppose that a request arrives and is blocked. Then, if the producer is in state s when the request arrives, the request will need to wait for $-s+1$ items to be produced before it can be unblocked (because there are $-s$ blocked consumers in line ahead of it). Let B_c be the time that a request spends blocked, given that it blocks. Then:

$$\begin{aligned} B_c = & \quad \frac{1}{\lambda} \sum_{s=0}^{M-1} (s+1) p(-s)/p_{mt} \\ = & \quad \frac{p(0)}{p_{mt}} \left(\left(\frac{p_b \mu_c}{\lambda}\right)^{M} \frac{M(p_b \mu_c - \lambda) - \lambda}{(p_b \mu_c - \lambda)^2} + \frac{\lambda}{(p_b \mu_c - \lambda)^2}\right) \end{aligned}$$

A request blocks only if all h_{max} producers that it probes are empty, which occurs with probability $p_{mt}^{h_{max}}$. Therefore:

$$B = p_{mt}^{h_{max}} B_c \tag{9}$$

We have now defined enough equations to solve the system. An explicit solution is infeasible, but iteration on μ_c works well. After the system of equations is solved, we can calculate the following performance measures:

- The average number of probes per request h_{avg}.
- The average time a consumer spends blocked $W = (h_{avg} + 1)r + p_{mt}^{h_{max}} B$.
- The utilization of the producers $U_p = 1 - p(f) = 1 - (\lambda/\mu_c)^f p(0)$.
- The utilization of the consumers $U_c = (1/\mu)/(1/\mu + W)$.
- The throughout $T = MU_c/\mu = NU_p/\lambda$.

4.1.1 Performance Comparison

We ran a set of experiments to validate the accuracy of the analytical model by comparing its results to the simulation results. In addition, we investigate several aspects of the performance of the stochastic distributed queue algorithm.

The performance of the stochastic distributed queue algorithm depends on the match between the total production rate and the total consumption demand. We define the *load* on the producers to be $l = M\mu/(N\lambda)$. A load of $l = 50\%$ means that the producers should be idle half of the time, and a load of $l = 200\%$ means that the consumers should be idle half of the time.

In the experiments, we set $\mu = \lambda = .01$. Sending a message requires 1 tick. There are 100 producers, each of which have five local buffers (i.e., $f_i = 5$). We varied the number of consumers to vary the load.

In Figure 2, we plot the average number of probes that the algorithm makes as we increase the maximum number of probes before blocking, for loads of 50%, 100%, 150%, and 200%. The points in the chart are simulation points, while the lines are computed from the analytical model. The chart shows that the analytical and simulation models are in close agreement. If the load is 100% or less, then only a few probes are used on average (less than 2). If the load is greater than 100%, then the h_{avg} grows in proportion to h_{max}.

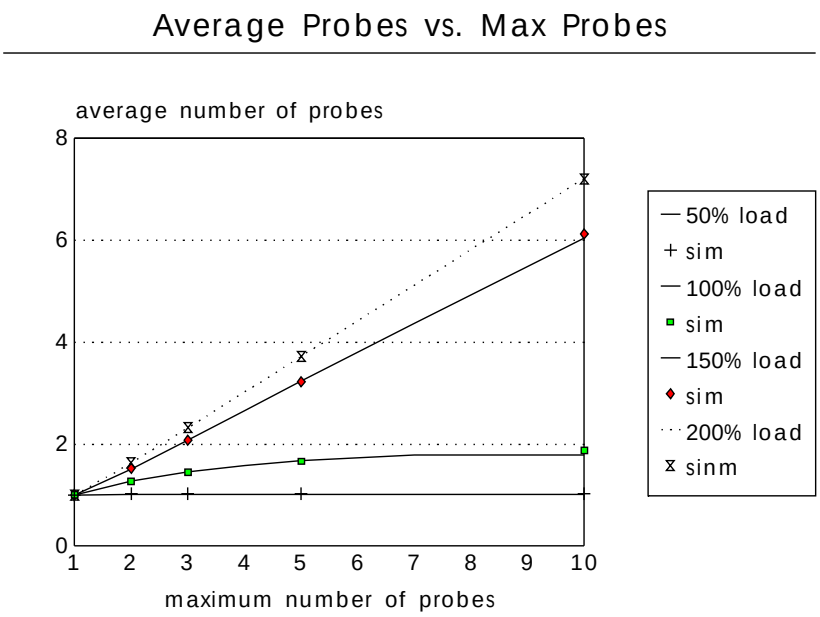

Figure 2: Average number of probes vs. maximum number of probes.

In Figure 3, we plot the average time that a consumer waits between issuing its request and receiving an object to consume (i.e., W). The points on the chart are simulation points, while the lines are drawn from

the analytical model. Again, both models have good agreement. The line for the 200% load does not extend to $h_{max} = 1$ because the model would not converge for this point. If the load is 100% or less, then the waiting time quickly approaches 2 ticks. If the load is greater than 100%, the waiting time quickly approaches a limiting value, of approximately $M/(N\lambda) - 1/\mu$. In all cases, the waiting time is close to its asymptotic value when h_{max} is 3 or greater. If the load is close to 100%, an additional performance improvement (of approximately 2.8%) can be gained by setting $h_{max} = 5$. Doing so can result in a significant increase in message passing overhead if the load is high. However, in the centralized queue manager algorithm, four control messages are sent for every object consumed. Figure 2 shows that even in a high load case, setting h_{max} to 5 means that $h_{avg} < 4$.

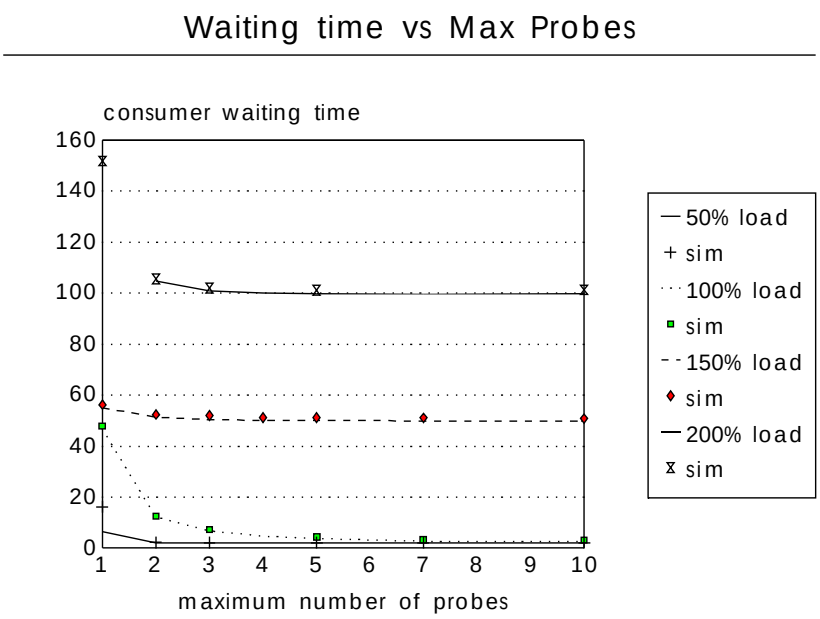

Figure 3: Waiting time vs. maximum number of probes.

The efficiency of the stochastic distributed queue algorithm can be seen by examining p_{mt}. We plot p_{mt} against the maximum number of probes in Figure 4. This chart contains data from the analytical model only, the points in the graph serve only to help identify the curves. As was suggested by the plot of W against h_{max}, most of the benefit of increasing h_{max} is achieved when $h_{max} = 3$.

In the previous three charts, we used 5 buffers at each producer. The number of buffers at each producer has a significant effect on performance, because increasing the number of buffers reduces the chance that a producer will block. In Figure 5, we plot the producer utilization against the number of buffers at each producer. For these experiments, $h_{max} = 5$. Most of the performance benefit of increasing the number of buffers is gained with using 5 buffers at each producer.

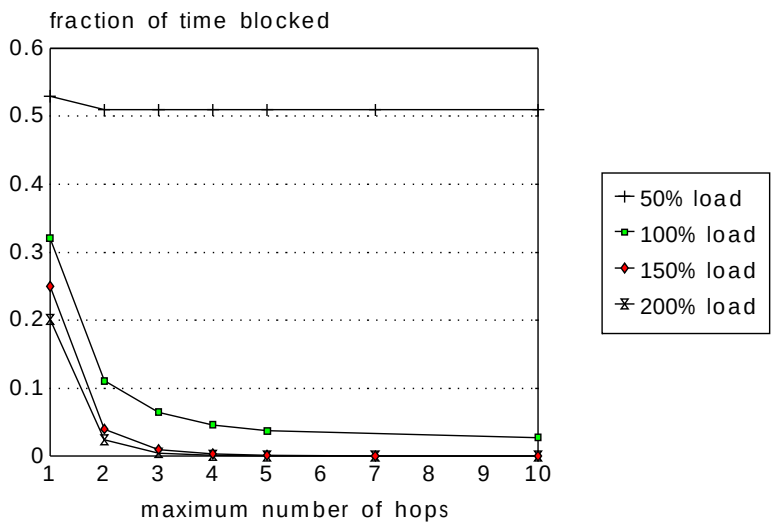

Figure 4: $p(f)$ vs. maximum number of probes.

Finally, we ran a simulation experiment to test how critical is the assumption of an exponential distribution for the production times. We ran a simulation in which the time required to produce an object is uniformly distributed in $[50, 100]$. We plot the analytical and simulation waiting times in Figure 6. The analytical model produces qualitatively accurate results, but is somewhat pessimistic. The simulation waiting times are lower because the variance in the production time are significantly lower in the simulation.

4.2 Non-homogenous Producers

The homogenous producers model reveals much about the performance of the stochastic distributed queue algorithms. However, the assumption that $\lambda_i = \lambda$ for every producer i is often not realistic. For example, the underlying computers might be heterogeneous. In this section, we will allow the producers to have different production rates, but we will still require that $p\_access_i = p\_access_j$ for every consumer i and j.

We assume that the producers are partitioned into K types, and every producer in type k has the same production rate λ_k. In addition, we assume that $p\_access[k] = p\_access[k']$ if producers k and k' are of the same type.

Of the N producers, N_k are of type k. The proportion of the request stream received by all type k producers is

$$frac_k = \sum_{i \text{ of type } k} p\_access[i] \tag{10}$$

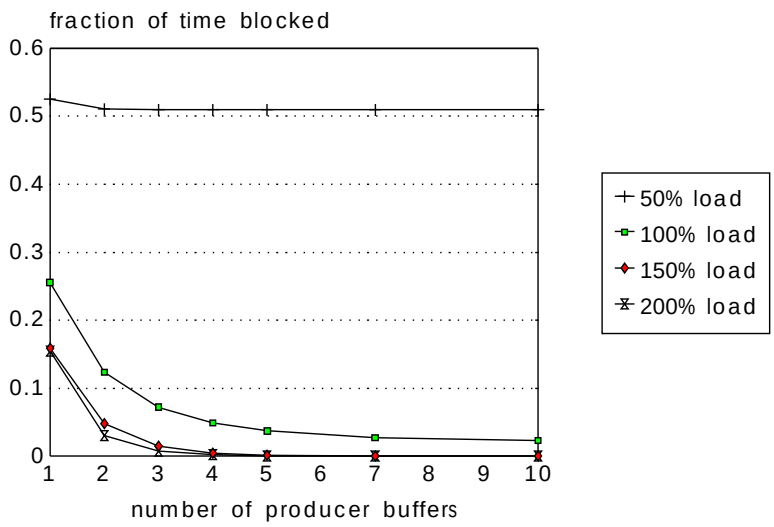

Figure 5: $p(f)$ vs. the number of buffers at each producer.

The arrival rate of requests to a type k producer is:

$$\mu_k = M * frac_k \mu_c / N_k$$

where μ_c is calculated using the formula 1. The per-type quantities $p_k(0)$ and $p_{mt,k}$ are calculated using the appropriate analogues of formulae 5 and 6. The probability of blocking p_b and the average number of probes h_{avg} depend on whether a request finds a random producer empty. The formulae for p_b and h_{avg} are the same as formulae 8 and 7 as long as we calculate the average probability of finding a producer empty, p_{mt}, which is:

$$p_{mt} = \sum_{k=1}^{K} frac_k * p_{mt,k} \tag{11}$$

The average time spent blocked at a producer of type k, given that a request blocks at a consumer of type k, $B_{c,k}$ is computed using the appropriate analogue of the formula in the previous section. The average time that a request spends blocked, given that it blocks, is a weighted average over $B_{c,k}$:

$$B_c = \sum_{k=1}^{K} frac_k * p_{mt,k} * B_{c,k} \tag{12}$$

By using these modifications, we again have enough equations to solve the system by using iteration.

4.2.1 Performance Comparison

We divided the producers into two groups: fast and slow. In the first set of experiments, the fast producers constitute 50% of the producers, and have twice the production rate of the slow producers (thus 67% of

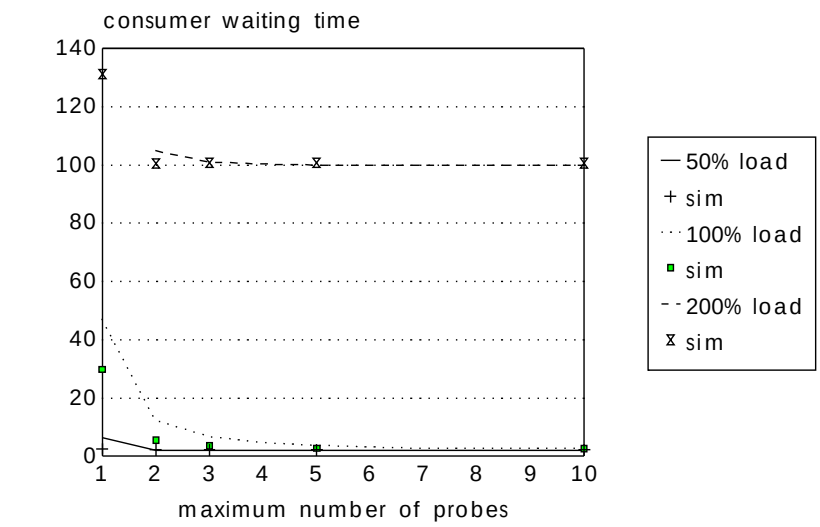

Figure 6: Waiting time vs. h_{max}. Object production times are uniformly randomly generated in the simulator.

the objects are produced by 50% of the producers). In the second set of experiments, the fast producers constitute 10% of the producers, and are 9 times as fast as the slow producers (thus 50% of the objects are produced by 10% of the producers).

In Figure 7, we plot the waiting time W against h_{max} for both sets of experiments. The points on the chart are simulation results, so this plot also serves to validate the analytical model. In this chart, $p\_access[i] = p\_access[j]$ for every pair of producers i and j (i.e., every producer receives the same request rate). In the moderately unbalanced (67/50) experiment, most of the performance benefit of increasing the maximum number of probes is achieved with $h_{max} = 3$. For the highly unbalanced (50/10) experiment, even a maximum of 10 probes does not achieve good performance. Since the fast producers make up only 10% of the producer population, there is a 35% chance that a consumer will not find a fast producer after 10 probes. In both cases, the simulation results are close to the analytical results, although the analytical model is somewhat pessimistic in the highly unbalanced case. The analytical model did not converge for $h_{max} = 1$, so we present only simulation results for that case.

In Figure 8, we plot the waiting time of a consumer against the fraction of requests directed to the fast consumers. In this chart, $h_{max} = 3$. As is expected, the waiting times are lowest when the fraction of consumer requests that are directed to the fast producers is proportional to the fraction of objects they produce.

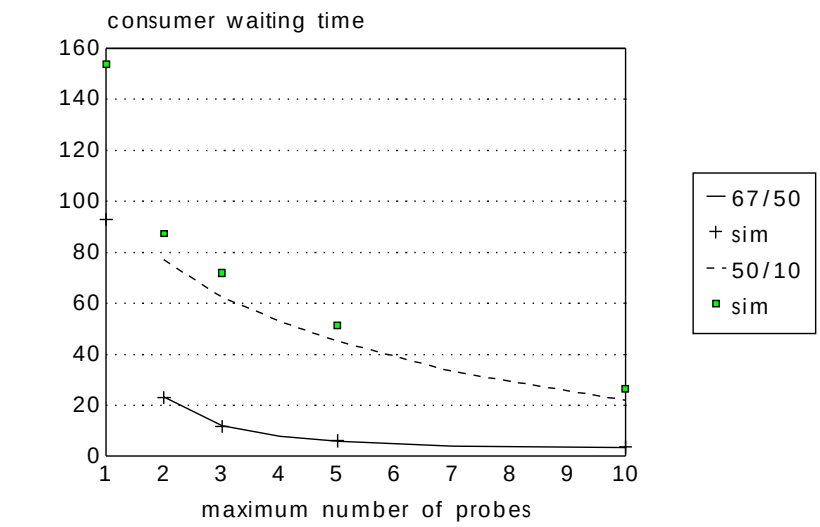

Figure 7: Waiting time vs. h_{max}. Fast and slow producers are equally likely to be probed.

4.3 Non-Homogenous Requests

In this section, we remove the restriction that p_access is uniform among all consumers. Unfortunately, this removes the simplifying assumption that all consumers are identical, and can be treated as a group. So, we will need to define many of the variables on a per-consumer basis.

We define B_i to be the average time that consumer i's request spends blocked, and $h_{avg,i}$ to be the average number of hops made by consumer i. Then the effective request rate issued by consumer i is

$$\mu_{eff,i} = \frac{h_{avg,i}}{1/\mu_i + (h_{avg,i} + 1)r + B_i} \tag{13}$$

The arrival rate of requests at producer i is

$$\mu_{c,j} = \sum_{i=1}^{M} p\_access_i(j)\mu_{eff,i} \tag{14}$$

Given the production rate at producer j, λ_j, the arrival rate of requests $\mu_{c,j}$ and the yet to be calculated probability of blocking $p_{b,j}$, we can calculate the state occupancy probabilities $p_j(s)$ and the probability that the producer is empty $p_{mt,j}$. Let M_j be the number of consumers that can send a request to producer j. That is, $M_j = |\{i|p\_access_i[j] \neq 0\}|$. If M_j is small, then blocking a consumer has a large effect on the request arrival rate, so $p_j(s)$ should be calculated based on equation 3 instead of equation 4.

Given $p_{mt,j}$ we can compute the probability that consumer i's probe finds a producer empty:

$$p_{mt}(i) = \sum_{j=1}^{N} p\_access_i(j) * p_{mt,j} \tag{15}$$

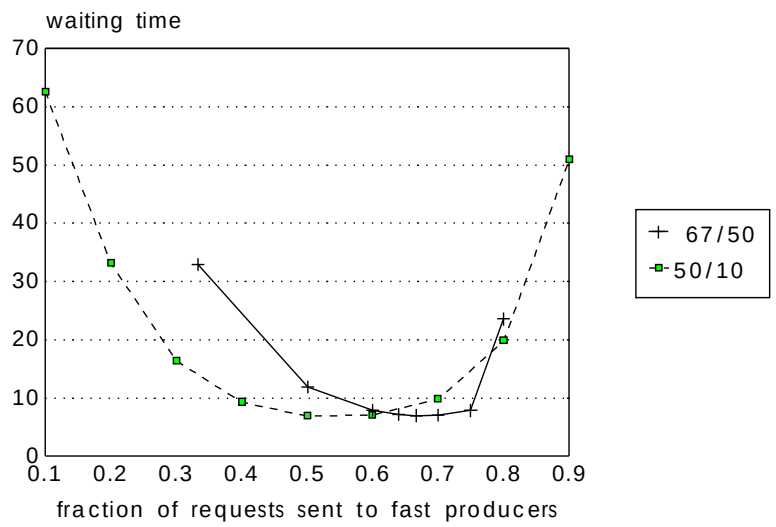

Figure 8: Waiting time vs. $frac_{hi}$. $h_{max} = 3$.

Given $p_{mt}(i)$, we can calculate the average number of hops made by consumer i's request, $h_{avg,i}$ and the probability that a request from consumer i will block, $p_b(i)$, by substituting $p_{mt}(i)$ for p_{mt} in formulae 7 and 8. Given $p_b(i)$, we can calculate the probability that a request blocks at a producer, $p_{b,j}$ by

$$p_{b,j} = \frac{\sum_{i=1}^{M} p\_access_i(j)\mu_{eff,i}p_b(i)}{\sum_{i=1}^{M} p\_access_i(j)\mu_{eff,i}} \tag{16}$$

Given the state occupancy probabilities of producer j, we can calculate the length of time that a request blocks, given that it blocks, by using the formula from the previous section. We can calculate the average time that a request from consumer i spends blocked by :

$$B_{c,i} = \sum_{j=1}^{N} p\_access_i(j) * p_{mt,j} * B_{c,j} \tag{17}$$

Finally we can calculate B_i by

$$B_i = p_{mt}(i)^{h_{max}-1} B_{c,i} \tag{18}$$

4.3.1 Performance Comparison

We executed an experiment to test the effect of limiting the number of producers a consumer can probe. We used 100 producers and consumers with identical production and consumption rates of $\mu = \lambda = .01$, and varied both the maximum number of probes h_{max} and the number of producers that each consumer is allowed to probe M_j. When consumer j chooses a producer to probe, it selects one of the M_j producers,

each with equal probability. In Figure 9, we plot waiting time of a consumer against M_j for different settings of h_{max}. We find that restricting M_j to any value larger than h_{max} has little effect on performance.

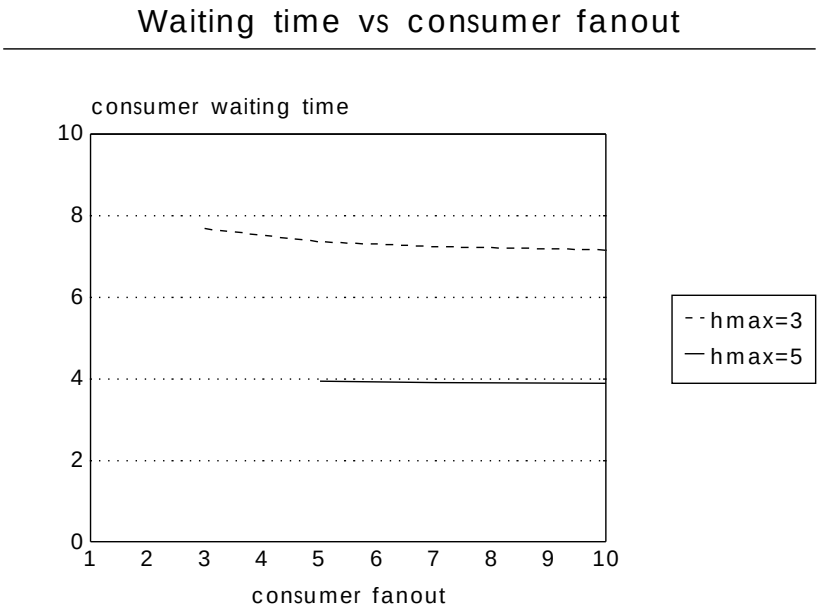

Figure 9: Waiting time vs. M_j.

5 Conclusions

In this paper, we present a simple stochastic algorithm to implement a distributed queue. The algorithm has several parameters, including the maximum number of probes a consumer makes before blocking, h_{max}, the producers that consumer j will probe, p_access_j, and the number of buffers at each consumer f. We develop a validated performance model of the stochastic distributed queue, and execute a performance study. We find that:

- Setting h_{max} to 3 works well for most loads and for moderately unbalanced producers.
- Setting h_{max} to 5 can increase performance by about 3% over setting h_{max} to 3, and still require fewer messages than the centralized manager algorithm.
- Using 5 buffers at each producer will give good performance. Additional buffers can give a small additional throughput improvement.
- The setting of p_access should be matched to the producer rates. A system with very unbalanced producers will not give low waiting times even with a large value of h_{max} unless p_access and the production rates are mathed.

- The number of connections between producers and consumers can be limited to $M_j > h_{max}$ with only a small degradation in performance.

References

[1] G.R. Andrews. *Concurrent Programming Principles and Practice.* Benjamin/Cummings, 1991.

[2] G.R. Andrews. Paradigms for process interaction in distributed programs. *ACM Computing Surveys*, 23(1):49–90, 1991.

[3] P. Avery, C. Chegireddy, J. Brothers, T. Johnson, J. Kasaraneni, and K. Harathi. The ufmulti project. In *Int'l Conf. on Computing in High Energy Physics*, pages 156–164, 1994.

[4] N. Carriero, D. Gelernter, and J. Leichter. Distributed data structures in linda. In *Proc. ACM Symp. on Principles and Practice of Programming Languages*, pages 236–242, 1986.

[5] R. Finkel and U. Manber. DIB - a distributed implementation of backtracking. *ACM Transactions on Programming Languages and Systems*, 9(2):235–256, 1987.

[6] L. George. A scheduling strategy for shared memory multiprocessors. In *Int'l Conf. on Parallel Programming*, pages I:67–71, 1990.

[7] S.F. Hummel and E. Schonberg. Low-overhead scheduling of nested parallelism. *IBM Journal of Research and Development*, 35(5):743–765, 1991.

[8] A.R. Karlin and E. Upfal. Parallel hashing: An efficient implementation of shared memory. *J. ACM*, 35(4):876–892, 1988.

[9] D. Kotz and C.S. Ellis. Evaluation of concurrent pools. In *Proc. Int'l. Conf. on Distributed Computing Systems*, pages 378–385, 1989.

[10] P.N. Lee, Y. Chen, and J.M. Holdman. DRISP: A versitile scheme for distributed fault-tolerant queues. In *Proc. Int'l Conf. on Distributed Computing Systems*, pages 600–607, 1991.

[11] U. Manber. On maintaining dynamic information in a concurrent environment. *SIAM Journal on Computing*, 15(4):1130–1142, 1986.

[12] F.J. Rinaldo and M.R. Fausey. Event reconstruction in high energy physics. *Computer*, pages 68–87, 1993.

[13] M. Sullivan and D. Anderson. Marionette: A system for parallel distributed progamming using a master/slave model. In *Proc. 9th Int'l Conf. on Distributed Computing Systems*, pages 181–187, 1989.

[14] *UFMulti Distributed Toolkit.* http://www.phys.ufl.edu/∼ufm.

[15] E. Upfal and A. Wigderson. How to share memory in a distributed system. *J. ACM*, 34(1):116–127, 1987.

[16] M.T. Vandevoorde and E.S. Roberts. Workcrews: An abstraction for controlling parallelism. *International Journal of Parallel Programming*, 17(4):347–366, 1988.