CS5330 Algorithm Experiment - Backoff Protocol

Xu Chen (A0144702N)

February 23, 2018

1 Design

The experiment investigates the performance of various backoff protocols in contention resolution with different arrival models, among multiple identical devices. For the ease of simulation, the experiment models a collection of devices trying to send a message without taking into account network communication, with discrete timestamps starting from 0. Each device has the below member fields that are present in all protocols (Protocols specific member fields will be included as necessary):

- *id*: A strictly increasing identifier for each device.
- W: The current window size for backoff.
- *num_retries*: The number of retries the device has made.
- t: Time unit that the device will try to send again.

A resource manager will poll each device at every single timestamp to check if the device wishes to send. If more than 1 device tries to send in the same timestamp, they all will backoff. A device that successfully sends will remain inactive. When a device backoffs, it will first obtain its new W based on a specific backoff algorithm, and randomly chooses a time unit in the newly determined window size W, and set t to be that time unit.

1.1 Randomization

In order to achieve fair benchmarking results, it is important to have an acceptable random number generator in choosing the time unit t when the device backoffs. Otherwise, a less randomized choice of t may produce results that do not represent the overall performance of an algorithm. The very first attempt is to use the $srand()$, and $rand()$ function calls in the standard C++ libraries. A very naive and common approach will be using the system timestamp as the seed for each run of the program. However, since the experiments will be run multiple times to sample the metrics, consective runs actually produce the same sequence of t. This is due to the implementation of $srand()$ and $rand()$ inherently, where casting of the timestamp incurs rounding errors when passed into the $srand()$ function. Consecutive runs with short time intervals will have the same seed value.

A better way, and also the implemented way of generating random t in this experiment, is to use the new $mt19937$ class from C++11 to obtain a random number. And at each call to get the t time unit, the device uses this run-specific random number as a parameter for the $uniform\_int\_distribution$ in the range of $[currentTime, currentTime + W]$.

1.2 Performance Metrics

The experiment uses the *time it takes for all devices to send* as the primarily metric when comparing multiple algorithms under different arrival models. In addition, the experiment also looks at the following 3 results:

- *The average number of tries each device made.* This might be crucial for devices which have a large overheads in retrying.
- *The average time taken between the arrival and finished time of the device.* This could be a good indicator of the latency for the system.

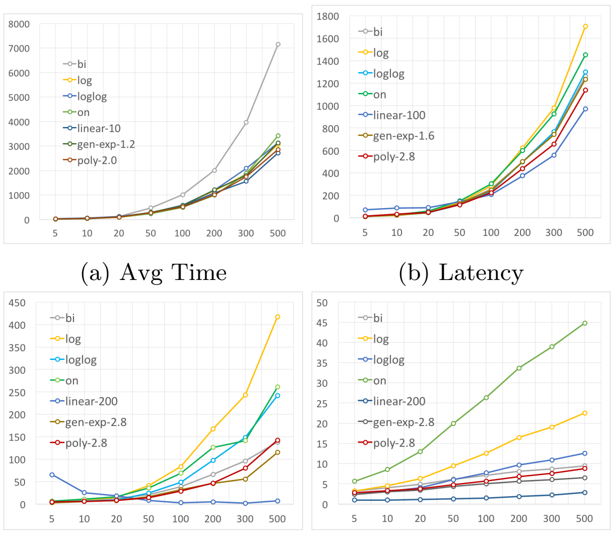
- *The time taken for one device to succeed.* It could be the interest of optimization in use cases such as leader selection.

For a particular experiment, it will be sampled 30 times with a maximal running time of $2e4$ time units. Any experiment that runs beyond that will be killed immediately (This is to prevent the progra runs forever in more bursty arrival models). For the 30 sampled results of a particular experiment, a 80 percentile will be used for comparison and charting. As the variance of the results is large due to randomization, the average value could be extremely skewed due to a couple of outliers. Therefore, an arbitrary 80 percentile is chosen to reduce the effect of that. In addition, it gives an indication of how bad an algorithm can be in most of the cases. (Thus almost like a worst-case condition).

2 Backoff Protocols

The first section of the experiment looks at how the system performs with various protocols in sending messages when no device arrive after the beginning. N devices will start with initial window size 1 at time 0.

- **Linear backoff**: on every retry, set $W = W + c$, for some small constant c.
- **Binary Exponential backoff**: On every retry, double the window size: set $W = 2W$. Thus if the window starts out as size $W = 1$, then after k tries, it has a window of size 2^k.
- **General Exponential backoff**: On every retry, set $W = W$ for some constant $\alpha > 1$.
- **Log backoff**: On every retry, set $W = W(1 + 1/\log(W)$.
- **Log-log backoff** : On every retry, set $W = W(1 + 1/\log\log(W))$.
- **Backon backoff** : Begin by setting $k = 1$. If $W > 1$, then set $W = W/2$. Otherwise, if $W = 1$, then set $W = 2^k$, and set $k = k + 1$.



(a) Avg Time (b) Latency

(c) First Success (d) Retries

Figure 1: Performace with no arrivals

- **Polynomial backooff**: After r tries, set $W = (r + 1)^\alpha$for some constant α.

Figure 1 shows the performace of various backoff protocols as number of devices N increases. For *Linear Backoff*, *General Exponential*, *Polynomial Backoff*, the experiment also tried with multiple parameters, and show the best results among them in the graph for comparison.

Average time to finish

From 1a, The Binary Exponential Backoff algorithm took the longest time to finish as number of devices increase. In addition, as N becomes larger, the performance of Binary Exponential Backoff decreases dramatically. This could be due to large window size as more retries occur since the window size W will increase exponentially. A trailing device with large window size may end up sending the device extremely late even though the channel has been mostly free. The rest of backoff protocols have similar performance while the Linear Backoff with small additional window size performs relatively better, followed by Polynomial Backoff with α being 2.0.

Latency

As for latency, on the contrary, Binary Exponential is not too bad, which also adds to the possibility that a few trailing devices might be the reason that it behaves so badly with average time. In terms of the best performing algorithm, Linear Backoff seems to work the best as number of devices increase, while Logarithm Backoff is the worst, which implies that a large number of devices might be in collision with each other until a very late stage when the window size W becomes large enough.

First Successful Send

From 1c, Linear Backoff performs the best, and as the number of devices N becomes closer to the additional window size at each back off, the first success becomes extremely fast. This fits the intuition that when the window size becomes closer to the numebr of devices, the collision will be less likely. On the other hand, the Logarithm Backoff behaves poorly, which also coorespondes to intuition as with the Logarithm Backoff, the window size increases more slowly.

Retries

It is not surprising that the Backon Backoff protocol has the worst performance in terms of number of retries, as when a collision happens, it will further decrease its window size W in most of the case, which results in an even higher chance of collision. And large additional window size (200) with Linear Backoff still performs the best. Devices only need to retry in average 4 times to get the message sent.

Therefore, this experiment shows that for different use cases, different protocols have huge difference in performance. With different things to optimize, different protocols should be used. For example, Linear Backoff with large additional window size at each backoff might be the best candidate for a leader election protocol or when retries overheads are high. It might be surprising that the Binary Exponential Backoff algorithm behaves rather badly in this experiment. This could be due to the specific setup of the expriment, where there are only fixed number of devices since beginning, and no devices arrive afterwards. This conjecture will be tested in the later sections.

3 Arrival Models

The previous experiment was based on the assumption that no devices join when the process begins. However, this is usually not true in reality. In this section, we tested the algorithms under multiple arrival models:

- **Bernoulli Arrival Mode**: In each time step, one new device arrives with probability p.
- **Normal Arrival Mode**: In each timestep,the probability distribution on the number of devices that arrive is Normal.
- **Square-root Batch Arrival Mode**: In each time step, with probability $1/\sqrt{n}$, a cluster of $\sqrt{n}$ new devices arrive. Otherwise, with constant probability p 1 device arrives.

3.1 Experiment Setup

In general, more devices arriving will defintely result in poorer performance for any kind of backoff protocols since more collisions are possible. In fact, when the rate of arrival is greater than the rate devices sending messages successfully, the experiment runs non-stop. Therefore, for various arrival models, the parameters to each distribution are important. In this experiment, we tuned the distribution so that for every 10 second, there are roughly the same number of new devices joined (3 to 3.5 every 10 seconds), with the exception of the square-root batch arrival model.

- Bernoulli Distribution: $p = 0.3$.
- Normal Distribution: $\mu = 0, \sigma = 0.7$.

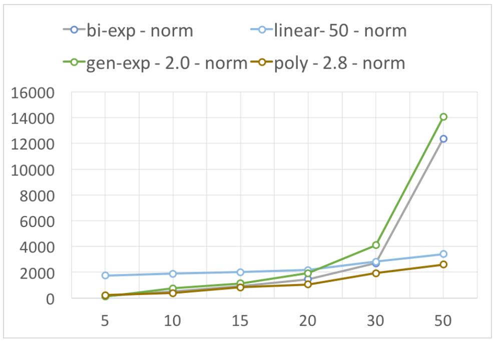
- Square-root Batch: $p = 0.001$. (In fact, this distribution is so bursty that it will have at least 10 new devices joined every 10 seconds).

Moreover, in order not to flood the report with too many charts and findings, only the average time until all devices finished metric will be presented in this section.

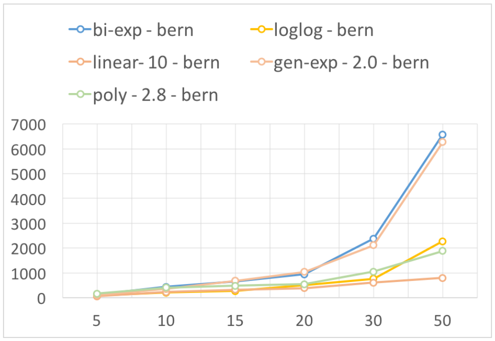
3.2 Results

As shown in Figure 4, some of the algorithms exceed the time limit allowed by the program, and that's why they are not shown in the graph. In general, General Exponential Backoff and Linear Backoff behaves relatively better than many of the counterparts. As for the Linear Backoff, its performance deteriorates rapidly when the additional window size is smaller than the number of devices.

As for the Square-root Batch Arrival Mode, all of the backoff algorithms are not able to finish within the maximal time limit, and therefore it is not shown in the results.



(a) None Arrival Mode



(b) Bernoulli Arrival Mode

Figure 2: Performance of Algorithms With Different Arrival Mode

4 Adaptive General Exponential Backoff

4.1 Hypothesis

From the previous section, many protocols (given by the Professor) were tested under various cases. While doing the experiment, a pattern observed from the log caught my attention.

There will be many collisions at the beginning, and as more devices send the message successfully, there will be periods where the channel is free but no devices attempts to send as they now have a large window size W. Therefore, an intuition will be, for a fixed given number of devices, an ideal algorithm should increase the window size W rapidly when consecutive collisions are detected, and increase the window size W less dramatically since the window size has been big with multiple tries.

With some trial and errors, we found a particular function that decreases in the domain of $[0, \infty]$: $\log(1/(ax) + 1)$, where $a \approx 0.01$. (Figure 3)

Consider the General Exponential Backoff, if we use this function on current window size to obtain the α dynamically at every backoff, we will increase the window size W greatly when W is small, and proportionally less or even smaller when W becomes large.

However, there is one issue with this function, as W becomes larger, if the value of a is larger than 0.01, the function decreases dramatically and approaches 0 too quickly. This reduction is too much as the window size becomes too small, resulting more collisions. And if a is smaller than 0.01, while we will have a better ratio when W increases, the initial values the function returns might be too big, making window size W grows too fast. Therefore, if we can make a smaller as number of retries grow, intuitively, we can have a balance: not growing too fast at the begin-

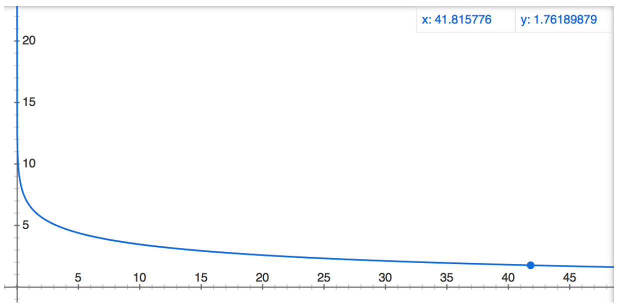


Figure 3: $\log(1/(0.01x)+1)$

ning and also not reducing too much as W increases.. With multiple trials, we arrived at this formula:

Adaptive General Exponential Backoff: At every backoff, multiple the current window size by α, where α is obtained from function $f = \log(\frac{1}{0.01/5(k+1)}W+1)$, k is current number of retries, W the current window size.

The multiplier 5 is arbitrary here with experiments that produce the best results. And the $k+1$ in the denominator is just to avoid division by 0 when the device just joins.

Adaptive Ratio

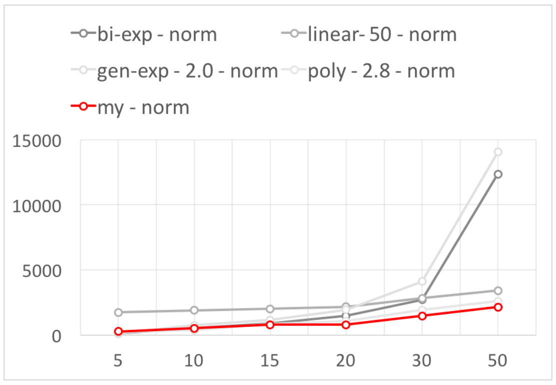
In addition, since the selection of t is random in the window size W, there is a chance that even the window size is big, the device might still try again with a relative small t. This might be bad. Consider the situation where there are many devices, and if a device re-tries too quickly, it will be more likely it gets collided with someone else again since others have not had the chance to send the message yet. However, the device will further increase its window size W even though the collision is actually due to an "unlucky" choice of t. Therefore, the intuition is to increase the window size W less if a device clash with others at an t that is very close to the previous retried time, and vice versa.

Adaptive General Exponential with Ratio Backoff: At every backoff, further multiple by $\beta = \log(2+\frac{t_i}{t_j})$, so that $W = \log(\frac{1}{0.01/5(k+1)}W+1)\log(2+\frac{t_i}{t_j})$, where t_i is the time since the last retried, and t_j is the time between now until the previous window ends.

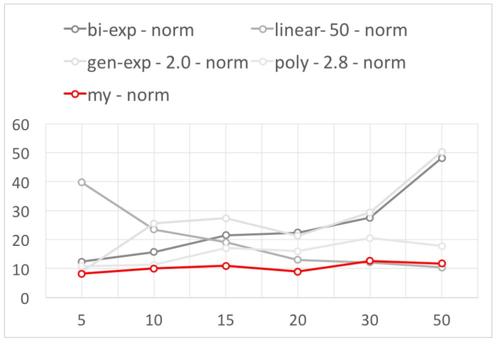
4.2 Experiment Results

With the above two changes to the General Exponential Backoff, we compared the performance of it with the rest under the Normal Arrival Mode. As shown in the Figure 4a, the algorithm performs better than the rest of the backoff algorithms.

In addition, the algorithm only incurs relatively small number of retries as well, making suitable for multiple purpose or use cases.



(a) Adaptive General Exponential with Ratio Backoff- Average time to finish



(b) Adaptive General Exponential with Ratio Backoff- Number of tries

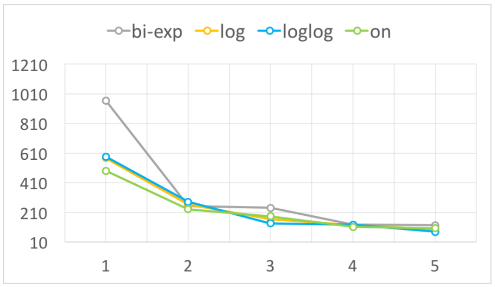
Figure 4: Performance of Algorithms With Different Arrival Mode

5 Power of Two

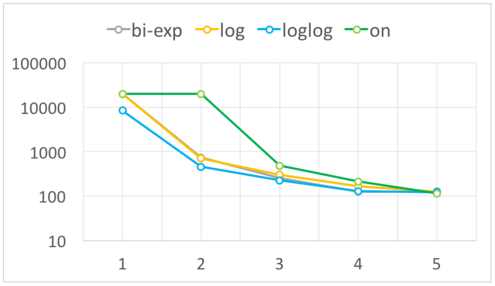
The "power of two choices" is one of the greatest findings for hashing in the recent years: each item is hashed to two buckets and placed in the one with the shorter linked list. Could the power of two also show its magical power in the case of backoff protocols?

More formally, for a given set of devices (and arrival model), how does the performance improve when there are multiple channels available? Intuitively, the improvement of the performance should be at least linear to the number of channels available M.

At each time unit, devices could send successfully as long as the number of devices requesting to send is no larger than the number of channels available. Otherwise, no devices can send, and they all have to backoff. This experiment focuses on the average time for all devices to send under the Normal Arrival Mode, and None Arrival Mode. The number of channels available will be from 1 to 5. The number of devices N at the beginning is 100.



(a) None Arrival Mode



(b) Normal Arrival Mode

Figure 5: Average time to send all with different number of channels

Experiment Result

As shown in the Figure 5a, when there are no devices arriving after the procedure begins, the effect of having multiple channels is not obvious. A constant factor of 2 to 3 reduction in time is observed with the addition of one channel.

However, when the number of devices arrive follows Normal Arrival Mode, the effect of having one more channel has is much significant, with 10 times more reduction. The magic power of two might be due to better collisions avoidance when new devices joined. When there are new devices joining the cluster, having one additional channel could effectively reduce the number of collisions happen since most of the time there will be only 1 device joined at each time unit under the Normal Arrival mode.

As the number of channels grow, the improvement becomes less significant for both arrival modes.

6 Power of Sharing

In the previous experiments, we assume a model where no communication is possible between the devices and the resource manager. On the other hand, we also know that collisions could be effectively reduced if the window size of a device W is appropriate so that the randomization will cause less collisions.

Therefore, it is natural to envision a model where the minimal communication is possible. (Of course, too much communication will just result in a centralized resource scheduler) For example, what if each device could know the number of devices present in the cluster then they first join? Or what if at each backoff, a device could know the number of devices in the cluster? We propose that with little information sharing, the performance of the system could be largely improved. This experiment aims to testify this conjecture. In particular, we aim to find out the effect of the below two sharing modes:

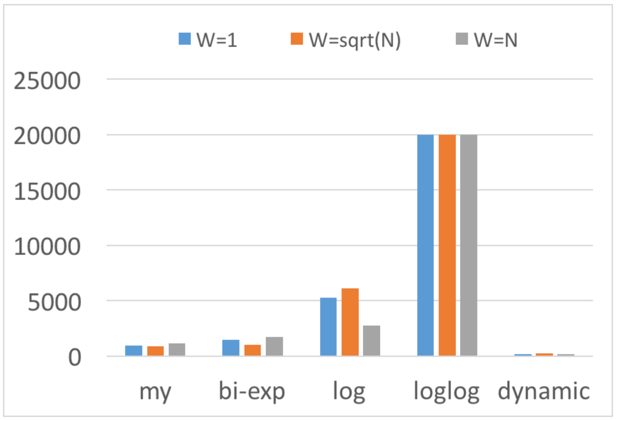


Figure 6: Communication Between Devices

- When a new device joins, it will set the initial window size W according to the current number of devices in the cluster. In this case, the new W could be 1, $sqrt(W)$, or W.
- When a device backoff, it will set its window size W according to the current number of devices in the cluster. We call this Dynamic Backoff

To achieve this, the experiment tests the Adaptive General Exponential Backoff, Binary Exponential Backoff, Log Backoff, and Log-log Backoff, as well as the Dynamic Backoff, under the Normal Arrival Mode.

Experiment Result

As shown in the Figure 6, setting the initial value of W according to the number of devices in the cluster has no observable effect under the Normal Arrival mode. This could be due to the fact that when W is set to 1, any new device joining the cluster will have a high probability finishing at the time unit it joins, while a larger W could mean a delay, which might result in more devices in the cluster.

On the other hand, setting the window size W according to the number of devices at each backoff however has significant effect on the performance that it is at least 5 times faster than my Adaptive General Exponential Backoff.