Erlang B and Engset formula calculation

Veljkovic Srdjan

November 2010.

1 Introduction

Erlang B formula calculates the probability of loss for offered traffic on a number of "lines" ("traffic carriers").

Pragmatically, Erlang B formula gives, for a number of (phone) calls which have to be transported via a number of voice paths (channels), probability that a call will fail (there will be no free channels).

Erlang B formula presumes an infinite number of sources for the traffic, and that a failed call will not influence the traffic. These presumptions are crude.

First presumption is appropriate when the number of sources is much larger than the number of channels (it is said "over 8 times larger", but we do not know the source of such claims). If this presumption doesn't hold, you should use Engset formula, detailed in section 3.

Second presumption is appropriate when the rate of call failure is low. If this presumption doesn't hold, you should use "Extended Erlang B formula", which is actually an algoritam, detailed in section 2.5.

For a detailed description of these terms you should read some book on traffic theory.

1.1 If there is B there must be C

Erlang C formula also exists, and is used if the failed call "is put on hold/wait", until a channel is released. The crude presumption here is that the call will wait "indefinetely", if it is not so, other methods/formulas are used.

We shall not deal with Erlang C formula further in this document.

2 Erlang B formula

Unfortunately, there are no general rules for naming the values in traffic theory, so we shall use "our own". Erlang B formula is:

$$p_g = \frac{\frac{s^k}{k!}}{\sum_{i=0}^{k} \frac{s^i}{i!}} \tag{1}$$

with following meanings:

p_g probability of call failure ("loss")

s offered traffic in Erlangs (number of simultaneous calls)

k number of channels to carry the traffic (calls)

2.1 Calculation

Erlang B formula uses factorial, so it can't be applied "as is" in todays computers for a larger k, because factorial for such numbers will be to big or to slow to calculate.

Also, Erlang B formula can be interpreted to give a relation between three parameters. It is often of interest to calculate one of the two input parameters of Erlang B formula, using the other input parametar and the result. If we know the offered traffic, we can determine the number of channels needed for that traffic for a given probability of loss. Also, if the number of channels is imposed by the environment, we can determine maximum possible traffic for a given probability of loss.

2.2 Recursive calculation of Erlang B formula

This calculation is widely known. The idea is that, for the same traffic, we can calculate the result for a number of channels if we know the results for a number smaller by 1. It is convinient to use the inverse formula in these calculations.

$$\frac{1}{p_g} = \frac{\sum_{i=0}^{k} \frac{s^i}{i!}}{\frac{s^k}{k!}} = \frac{k!}{s^k} \sum_{i=0}^{k} \frac{s^i}{i!} = \sum_{i=0}^{k} \frac{k! s^k}{s^i i!}$$

$$\frac{1}{p_g} = \sum_{i=0}^{k} \frac{k!}{s^{k-i} i!} \tag{2}$$

$$\frac{1}{p_g(k+1)} = \sum_{i=0}^{k+1} \frac{(k+1)!}{s^{k+1-i} i!} = \frac{(k+1)!}{(k+1)!} + \sum_{i=0}^{k} \frac{(k+1)!}{s^{k+1-i} i!} = 1 + \frac{k+1}{s} \sum_{i=0}^{k} \frac{k!}{s^{k-i} i!}$$

$$\frac{1}{p_g(k+1)} = 1 + \frac{k+1}{s} \frac{1}{p_g(k)} \tag{3}$$

Initial value - for $k = 0$, is calculated "directly" from 2:

$$\frac{1}{p_g(0)} = \frac{0!}{s^0 0!} = 1$$

This gives us the following algorithm in the Ruby programming language:

```
def erlangb(s, k)
    r = 1.0
    1.upto(k) { |j| r = 1 + r*j/s }
    return 1.0 / r
end
```

This is code to ilustrate the algorithm, not for "industrial" use.

2.3 Calculate number of channel by bisection

To calculate the number of channels, we use the fact that, for a constant traffic, p_g is a decreasing function (as k grows larger, p_g grows smaller). This is intuitive, for more channels, the losses are smaller. Yet, it can be proven.

Here we give the prof of Quiao and Quiao, in an article than can be found on the Internet:

`http://www.cas.mcmaster.ca/~qiao/publications/erlang/newerlang.html`

Quiao and Quiao give a different recursive Erlang B formula then "ours". There is no detailed description how the formula was generated, so we can't tell if there is some error, or the difference can be discarded in real world cases.

But, their proof that p_g is constantly decreasing is sound. We shall re-tell it here:

$$p_g(k+1) = \frac{\frac{s^{k+1}}{(k+1)!}}{\sum_{i=0}^{k+1}\frac{s^i}{i!}} = \frac{s}{k+1}\frac{\frac{s^k}{k!}}{1+\sum_{i=1}^{k+1}\frac{s^i}{i!}} < \frac{\frac{s^k}{k!}}{\frac{k+1}{s}\sum_{i=1}^{k+1}\frac{s^i}{i!}} = \frac{\frac{s^k}{k!}}{\frac{k+1}{s}\sum_{i=0}^{k}\frac{s^{i+1}}{(i+1)!}} =$$

$$= \frac{\frac{s^k}{k!}}{\sum_{i=0}^{k}(\frac{k+1}{i+1}\frac{s^i}{i!})} < \frac{\frac{s^k}{k!}}{\sum_{i=0}^{k}\frac{s^i}{i!}} = p_g(k)$$

Plainly :

$$p_g(k+1) < p_g(k)$$

Since we know that the function is decreasing, we can use the bisection method. All we need are two values of $p_g(k)$, one smaller and one larger than the given P_g - actually we need to values k for which that is true.

We shall start with 0 and the offered traffic as start values for k. Buf, if $p_g(s) < P_g$, we need such $k > s$ for which $p_g(k) > P_g$.

Quiao and Quiao suggest, without a sound reason (they simply state "we found it to be the best solution") to increase the traffic in steps of 32, until we find such k. We think that 32 is a "magic number" here, as it is unclear why it should be a good number for traffic greater than, say, 1000E. So we use a different method.

We think that it is better to use a factor, instead of a step, becaus a factor will make the calculation dependent of the offered traffic. We thing that the factor of 1.5 is good. The

losses have to be very low for you to need 50% more channels than the offered traffic. Such low losses are rarely needed, but if they are, additional steps will find the needed k.

We can now show the algorithm in the Ruby programming language.

```
def chan_erlangb(p_g, s)
    l, r = 0, s.ceil
    while erlangb(s, r) > p_g
        l = r
        r = 1 + (r * 3) / 2
    end
    mid = 0
    while (r-l) > 1
        mid = (l+r+1)/2
        if erlangb(s, mid) > p_g
            l = mid
        else
            r = mid
        end
    end
    return r
end
```

Again, this code is for illustration only.

2.4 Calculate max traffic by bisection

To calculate the maximum possible traffic, we use the fact that, for a constant number of channels p_g is increasing. This is intuitive, the higher the traffic, the higher the losses. Yet, this can be proven.

We shall use the method of first derivative. We shall prove:

$$p'_g(s) > 0 \iff k = const \tag{4}$$

Using the widely known equation:

$$f'(\frac{u}{v}) = \frac{f'(u)f(v) - f(u)f'(v)}{v^2}$$

on the Erlang B formula (1), presuming that k is constant, gives us:

$$p'_g(s) = \frac{\frac{ks^{k-1}}{k!}\sum_{i=0}^{k}\frac{s^i}{i!} - \frac{s^k}{k!}\sum_{i=0}^{k}\frac{is^{i-1}}{i!}}{(\sum_{i=0}^{k}\frac{s^i}{i!})^2}$$

For the purpose of our proof, observe that the denominator is a sqaure of some expressions, so it is ≥ 0, and can only be 0 if $s = 0$ (no traffic), which is of no interest to us. So we can analyse the sign of the nominator only.

$$\frac{ks^k}{k!s}\sum_{i=0}^{k}\frac{s^i}{i!}-\frac{s^k}{k!}\sum_{i=0}^{k}\frac{is^{i-1}}{i!}=\frac{s^k}{k!}(\sum_{i=0}^{k}\frac{ks^{i-1}}{i!}-\sum_{i=0}^{k}\frac{is^{i-1}}{i!})$$

Since $\frac{s^k}{k!}>0$, we can analyze only the expression inside the brackets:

$$\sum_{i=0}^{k}\frac{ks^{i-1}-is^{i-1}}{i!}=\sum_{i=0}^{k}\frac{s^{i-1}(k-i)}{i!}$$

Now it is clear that the sign depends only on:

$$\sum_{i=0}^{k}(k-i)=(k+1)k-\sum_{i=0}^{k}i=k(k+1)-\frac{k(k+1)}{2}>0$$

We can use the bisection method, much like what we used for calculating the number of channels, so we shall give the Ruby code without further explanations:

```
def traffic_erlangb(p_g, k)
    l, r = 0, k.ceil
    while erlangb(s, r) < p_g
        l = r
        r = 1 + (r * 3) / 2
    end
    mid = 0
    while (r-l) > 1
        mid = (l+r+1)/2
        if erlangb(mid, k) < p_g
            l = mid
        else
            r = mid
        end
    end
    return r
end
```

2.5 Extended Erlang B formula

If with p_d we designate the probability of an immediate re-attempt after call failure, then the algoritam of the Extended Erlang B formula is:

1. Calculate the Erlang B formula (1)
2. Add the following to the offered traffic: sp_dp_g
3. Calculate Erlang B formula with the offered traffic from the previous step
4. Repeat previous two steps until p_g becomes stable.

Here "becomes stable" means that the difference between two subsequent values of p_g is smaller than a measure of significance.

Algorithm in the Ruby programming language:

```
def erlangb_extended(s, k, p_d, epsilon)
    r, p_g = 0.0, erlangb(s, k)
    delta = epsilon
    while delta >= epsilon
        r = erlangb(s, k)
        delta = (r - p_g).abs
        p_g = r
        s += s * p_d * p_g
    end
    return p_g
end
```

In the code `epsilon` is the said measure of significance.

Calculation of the number of channels or the traffic from the extended Erlang B formula is basically the same as the Erlang B formula proper. The only difference is that, when calculating the Erlang B formula proper in the course of the algorithm, we shall use the p_d parameter. This is so obvious that we don't feel the need to provide further code or explanations.

3 Engset formula

From a certain point of view, Engset formula is a generalization of the Erlang B formula, which doesn't ignore the number of sources of traffic.

$$p_g = \frac{s^k \begin{pmatrix} v \\ k \end{pmatrix}}{\sum_{i=0}^{k} s^i \begin{pmatrix} v \\ i \end{pmatrix}} \tag{5}$$

A new value is added to the Erlang B formula (1): v - the number of call sources.

We can rearrange the formula:

$$p_g = \frac{\frac{s^k v!}{(v-k)!k!}}{\sum_{i=0}^{k} \frac{s^i v!}{(v-i)!i!}} = \frac{\frac{s^k}{(v-k)!k!}}{\sum_{i=0}^{k} \frac{s^i}{(v-i)!i!}}$$

It is now obviuos that the Erlang B formula and the Engset formula are very similar. That is why we shall use a similar procedure to devise a recursive formula:

$$\frac{1}{p_g(k)} = \frac{(v-k)!k!}{s^k} \sum_{i=0}^{k} \frac{s^i}{(v-i)!i!}$$

$$\frac{1}{p_g(k+1)} = \frac{(v-k-1)!(k+1)!}{s^{k+1}} \sum_{i=0}^{k+1} \frac{s^i}{(v-i)!i!} =$$

$$= \frac{(v-k-1)!(k+1)!}{s^{k+1}} \left(\sum_{i=0}^{k} \frac{s^i}{(v-i)!i!} + \frac{s^{k+1}}{(v-k-1)!(k+1)!} \right)$$

$$= \frac{(k+1)}{s(v-k)} \frac{(v-k)!k!}{s^k} \sum_{i=0}^{k} \frac{s^i}{(v-i)!i!} + 1 = \frac{(k+1)}{s(v-k)} \frac{1}{p_g(k)} + 1$$

This is similar to what we got for the Erlang B formula (3), but we shall rearrange it a little to make it easier to code:

$$\frac{1}{p_g(k+1)} = \frac{(k+1)}{s(v+1-(k+1))} \frac{1}{p_g(k)} + 1 \tag{6}$$

Like we did for the Erlang B formula, we shall calculate the initial value "directly":

$$p_g(0) = \frac{s^0 v!}{s^0 v!} = 1$$

Now it's easy to code this in Ruby:

```
def engset(s, k, v)
    r, v_1 = 1.0, v+1
    1.upto(k) { |j| r = 1 + r*j/(s*v_1) }
    return 1.0 / r
end
```

It is interesting that the popular method of Engset formula calculation is very different and doesn't solve the problem of large factorials. Maybe this is because the Engset formula is mostly used for small numbers, where this is not a problem. Still, we believe this to be a better algorithm since it permits use of Engset formula for a large number of sources/channels when there is high traffic per source. This is useful for analyzing "worst case scenarios" in system design.

Calculation of number of channels, sources and traffic is similar to those of Erlang B formula. Some additional info on those calculations may be inserted here, if the need arises.