CHAPTER 4

WEIGHTED AVERAGE SYNCHRONIZATION ALGORITHM

Chapter 4

Weighted Average Synchronization Algorithm

4.1. Overview

We present a synchronization algorithm, which achieves clock synchronization by using a weighted averaging as a convergence function. This algorithm is called Weighted Average Synchronization Algorithm (WASA) and it utilizes the concept of sliding window to find the minimum variance data set upon which the convergence function is used.

4.1.1. Distribution of Bad Clocks

In a network, each node exchanges its clock value and store in an array in order of their clock values. The clock values distributed in the array can be visualized in the following fashion: n clock values are distributed in the array i.e. we can see this as n slots to be filled with n-f good clock values and f bad clocks. The good and bad clocks values can be anywhere in the n slots. There are ${}_nP_f$ distinct possibility as depicted in figure 4.1 below. The two extreme cases are as shown where all the bad clocks have accumulated in the left or the right corner as shown below in figure 4.2.

4.1.2 Minimum Variance Window

The good clocks in a network spread around the ± δ from the mean if the numbers of bad clocks are just less than one third of the total clocks. Bad clocks can be anywhere in the array. As most of the world values are normally distributed, approximately sixty eight percent of these n clock values are within ± σ of the mean, where σ is the standard deviation of the distribution of

---

[1]The work presented in this chapter has been published in "**A precise clock synchronization algorithm in network**". Journal of communication engineering & systems. ISSN: 2321-5151.Volume 10, issue 1, 2020. **UGC CARE LIST JOURNAL (upto 2019)**

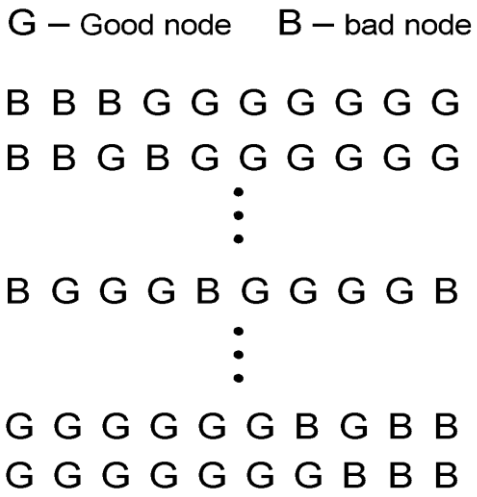

Figure 4.1: Distribution of good and bad clock values

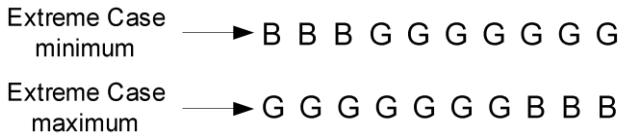

Figure 4.2: Extreme distribution of good and bad clock values

all the clocks. σ is a measurement of deviation of the values from the mean. Initially the window contains the left most clock values of the array as its left most member. We slide this window from the left a clock value ahead along the array. The sliding starts from left corner of the array and move towards right. We find the variance of the window at each window instant. The minimum variance window is the window containing clock values within ± σ from the mean. If the size of the window is reduced around two-third i.e., sixty six percent of the total clocks, then also the member clock values are same with few reductions in the number of clock values. This is particularly true if there are no malicious clocks present, which may give different clock values to different nodes. In presence of malicious clocks, the member of the window may change by up to f clock values. However, since the window must maintain minimum variance, the means calculated by the different arrays stored by different nodes vary within a bound. We use this concept in our algorithm.

In a network, two kinds of player are there with conflicting goals. One set of players, which have objective to stabilize the system, are the good nodes. The other set of players, which aim to destabilize the system, are the bad nodes. Our goal is to synchronize the network even in presence of bad nodes. The most serious types of bad nodes are those nodes,

which give inaccurate, untimely and conflicting information. They may also give different time to different nodes at the same time instance. These malicious behaviours of nodes can be of classified as follows: Consistent misbehaviour - Nodes exhibit the same malicious behaviour in temporal domain. Inconsistent misbehaviour - Nodes display malicious behaviour is inconsistent fashion in temporal domain. A node can misbehave in individual capacity or can collectively collude. These misbehaviours are categorized as follows: Individual misbehaviour: A node is displaying misbehaviour in individual manner without consulting other bad nodes. Collaborative malicious behaviour: Bad nodes may consult each other and decide their strategy in collective fashion.

In this thesis, our focus is to counter the individual misbehaviour, which includes consistent and inconsistent behaviour both. We have devised a sliding window technique to find the set of values, which show minimum deviation among the values. We know that Gaussian distribution is widely present in nature. We capitalized this knowledge to design the weight function. This weighted approach is used to mitigate the misbehaviour.

4.2. Description of the Algorithm

We give a detail description of WASA in this section. WASA works in round of equal intervals. Every good node in the network executes this algorithm. T_r time duration before the end of each round is called re-synchronization time. During re-synchronization period, each node executes WASA. The duration of a round is constant R, which is determined by the maximum allowed deviation range. Hence, R is decided according to the requirement of the system design. WASA can counter consistent malicious nodes. It also works in presence of inconsistent malicious behaviour provide the malicious nodes remain consistent during T_r. The figure 4.3 depicts the relationship of R and T_r.

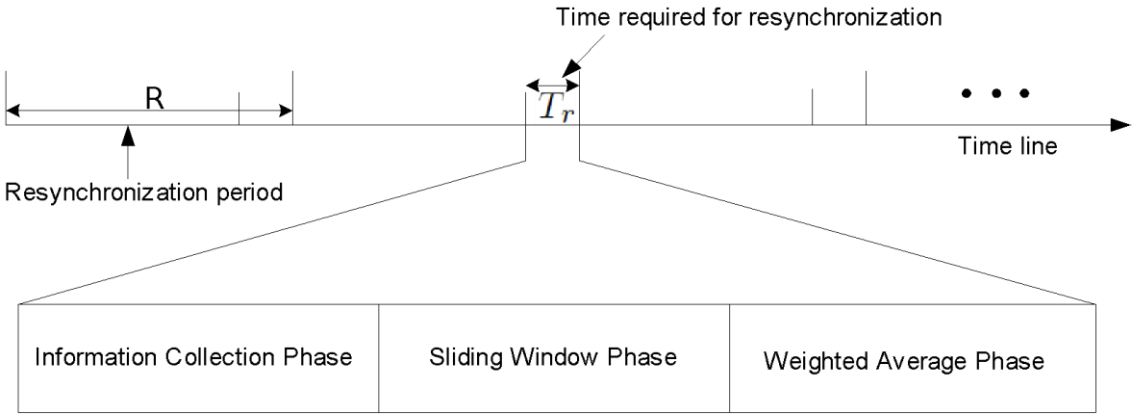

Figure 4.3: Working of a Clock and re-synchronization period

WASA consists of three phases namely Information Collection phase, Sliding Window phase and Weighted Average phase.

4.2.1. Information Collection phase

In this phase every nodes share their clock with one another. Node stores these clock values in an array. The clock values in the array are then arranged in an ordered fashion as per their values. We can visual these sorted clock values as points sitting the timeline.

4.2.2. Sliding Window phase

We introduced the notion of window here. The size of window is n-f where n network size and f are allowed bad nodes. Initially we placed the left boundary of the window at the beginning of the array. So, first window consists of first n-f values of the array. Thereafter we slide the window one step in the right direction and this consists of next n-f values. We continue this process until the right boundary of the window reaches end of the array. In this process we find the f+1 data set of size n-f. Now out of f+1 data sets we find out the data set with minimum variance. The variance of a random variable X is the the expected value of the squared deviation from the Mean of X, $\mu = E[X];$

$$Var(X) = E[(X - \mu)^2]$$

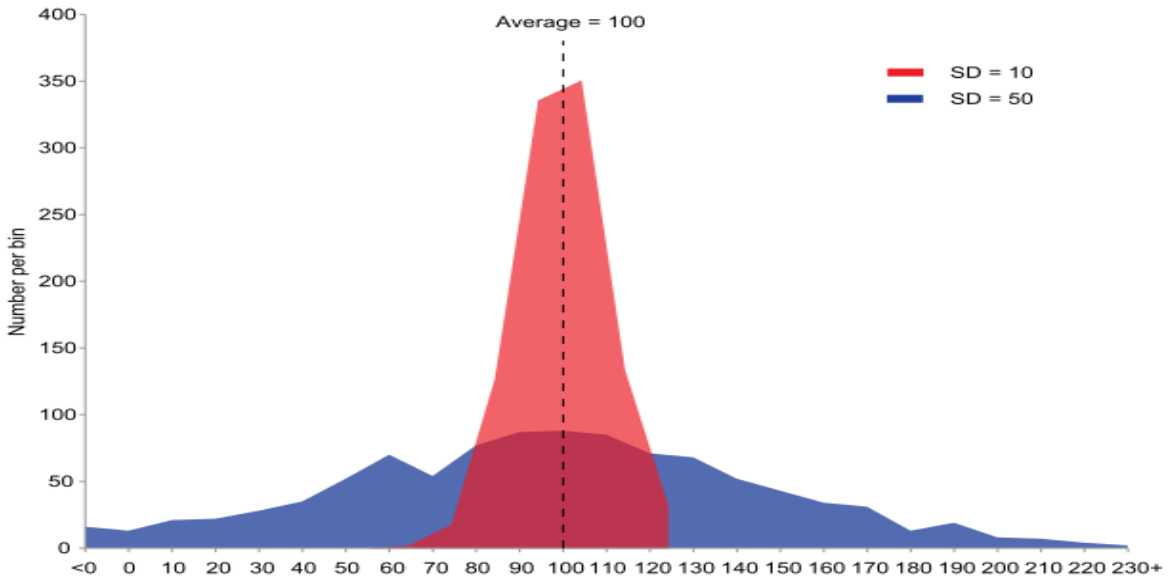

Fig 4.4 Graph of Variance (Diagram courtesy Wikipedia.org)

4.2.3. Weighted Average phase

Weighted Average or weighted arithmetic mean is similar to an ordinary arithmetic mean (the most common type of average), except that instead of each of the data points contributing equally to the final average, some data points contribute more than others. The notion of weighted mean plays a role in descriptive statistics.

$$W = \frac{\sum_{i=1}^{n} w_i X_i}{\sum_{i=1}^{n} w_i}$$

W = weighted average
n = number of terms to be averaged
w_i = weights applied to x values
X_i = data values to be averaged

We use the minimum variance data set calculated in the Sliding Window phase for finding the weighted average with some modifications. Since the size of the minimum variance window is n-f, it contains at least n-2f good clocks and at most f bad clocks. For the worst case, the minimum variance window contains n-2f good clocks and f bad clocks. There can be also at maximum f good clock outside the minimum variance window.

We now push those good clocks which are outside the minimum variance window inside the window by assigning the value of the good clock nearest to them inside the window. Now data set values inside the minimum variance window are assigned weights as per weight function and the weighted average is calculated. These weighted average value, θ , is the new clock.

For assignment of the weights, clock value within one $\pm \sigma$ distance from μ are assigned weight ω_1 , where μ is the mean and σ is the standard deviation of the window. Clocks within $\pm \sigma$ to $\pm 2 \sigma$ are assigned weight ω_2 and clocks within $\pm 2 \sigma$ to $\pm 3 \sigma$ are assigned weight ω_3. Finally, clocks beyond $\pm 3 \sigma$ are assigned zero weight. A pictorial description of working of WASA is given below at figure 4.4.

4.3. Theoretical analysis

In this section, we evaluate the performance of WASA algorithm. Initially all clocks are synchronized in some sense. At the start of the first synchronization period all the correct clocks are within α of the real time t (Initial synchronization). The distance between any two correct clocks is equal to or less than δ at the end of R. If ε is the maximum message delay then width of the window is at most $\delta + \varepsilon$. Now we discuss the performance of WASA for the ideal and worst case scenario.

4.3.1 Ideal Case Scenario

In the ideal case scenario there is no bad node in the network. Hence, the data sets of clocks are the same for all the clocks. The minimum variance of these windows is the same. Therefore, the weighted average, θ is identical across all the clocks.

Lemma 1. The precision of WASA in an ideal condition where there is no bad clock is $\pi = \varepsilon$, which is optimal.

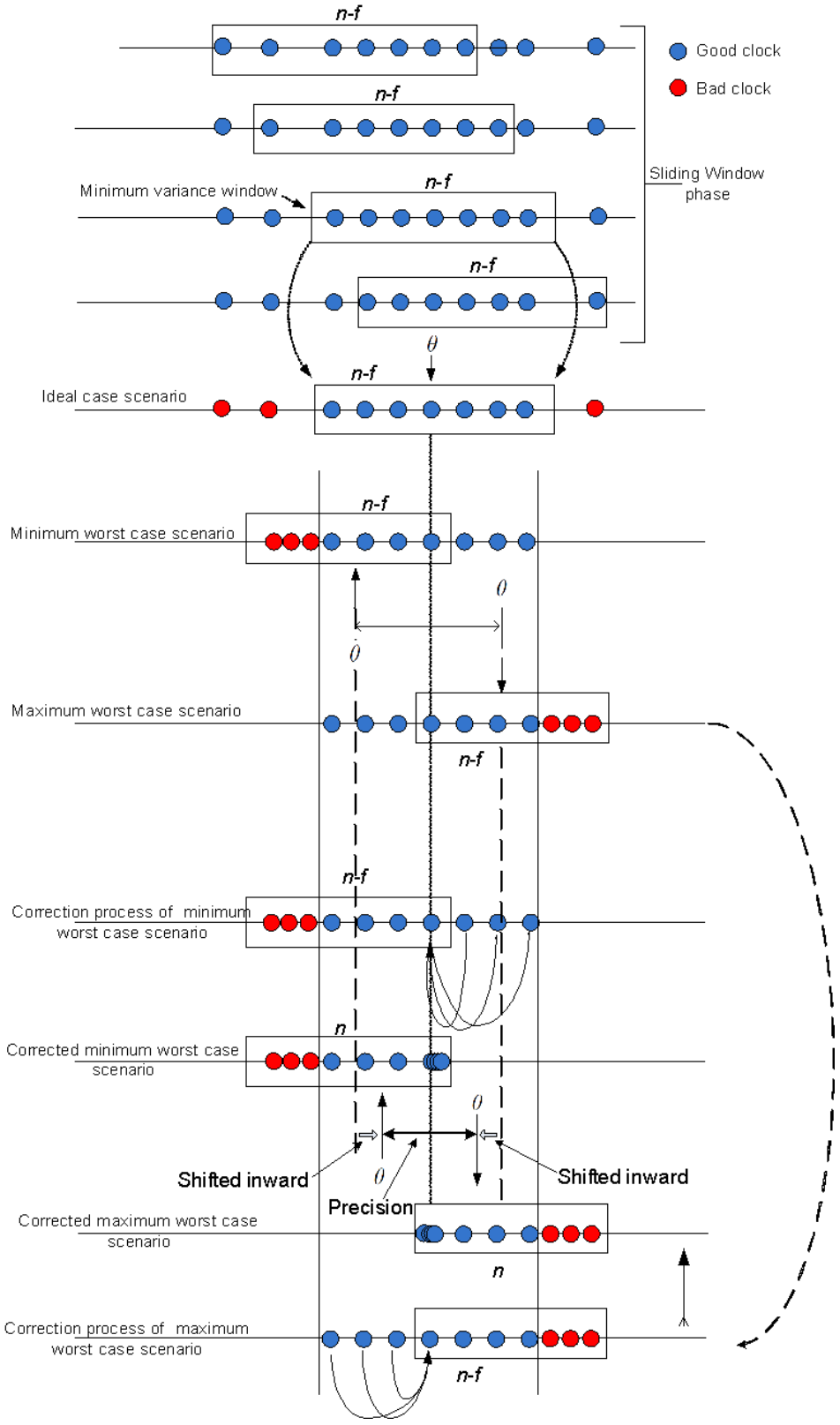

Figure 4.5: Pictorial description of working of WASA

Proof. The size of the minimum variance window is n-f but for a where case f=0, hence the size is now n. This means there are only one window and a single weighted average value, θ, which is the new clock. Hence the precision is optimal.

4.3.2. Worst Case Scenario for Proof of Correctness

The Here we do worst case analysis to establish correctness of WASA. We know that if an algorithm is validated for the worst case then it holds for all possible cases. The worst case is the condition when WASA gives the worst possible precision. This condition is obtained where all the bad nodes are malicious nodes. From Maximum Clock Failure assumption the size of the network is n=3f+1. Let C min and C max be the minimum and the maximum clock created by WASA immediately after R. Also let C_min window be the window from where C_min is calculated and similarly C_max window for C_max. Here the malicious clocks are all within the minimum variance window and make C_(min) the least and C_max the largest. This happens when all the malicious clocks are smaller than or equal to the smallest good clock. The good clocks have minimum drift and delay. In case of C_max all the malicious clocks are larger than or equal to the largest good clock. The drift and delay is the highest. A distribution pattern of the clocks for the worst case is depicted in the figure 4.5 below. As shown any good clock are in either of the windows. The C_min window contains f+1 good clocks out of which f good clocks are exclusively within it. Similarly the same distribution for C_max window. One good clock is common to both the windows. The difference between C_min and C_max gives the worst case precision, π, of WASA.

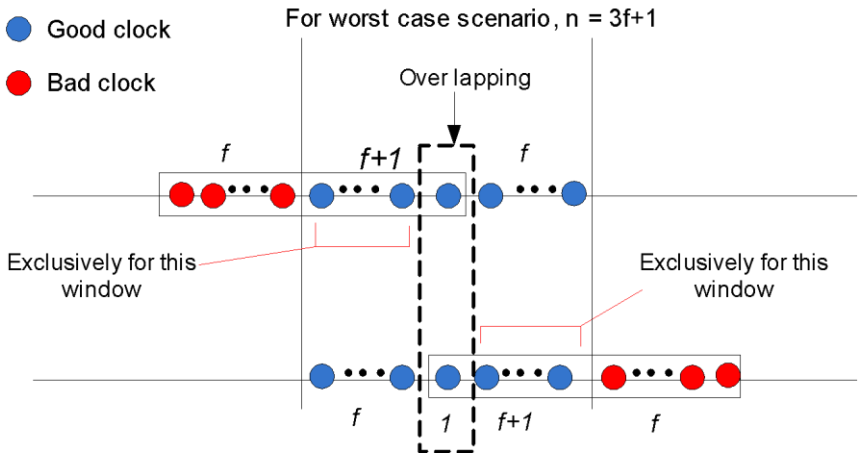

Figure 4.6: Clock values distribution for worst case scenario

Theorem 1. In the worst case the precision of WASA, , if the upper bound of bad clocks is one-third of the total number of nodes, n, with a maximum window width δ + ε is given by

$$\pi \leq \varepsilon + 2(\delta + \varepsilon)\left(\frac{f}{n}\right) = \varepsilon + 2(\delta + \varepsilon)\left(\frac{f}{3f+1}\right)$$

Proof. Here Π is the distance between C_{min} and C_{max}. From *Lemma 1* we know that for ideal condition all the clocks are within ε of θ. With the presence of bad clocks θ is shifted to either direction. In case of C_{min} all the malicious clocks is to the left of the left most good clock. The maximum deviation of $C_{min}window$, $\delta_{min} \leq \delta$ because it is the minimum variance window. C_{min} is obtain by shifting θ by an amount upto $f\frac{\delta}{(n-f)}$ since there are f malicious clocks. The shift is made lesser by pushing the left out good clocks into the window. Now the number of clocks is increased to n from n-f. Hence the distance shifted is up to $f\frac{\delta}{n}$.

Similarly, the shifting due to malicious clocks on the right side of the good clock is up to $f\frac{\delta}{n}$ on the right side of θ. This gives value of C_{max} w.r.t. θ. Hence considering ε, the distance between C_{min} and C_{max} is given by

$$\pi \leq \varepsilon + \left\{\theta + (\delta + \varepsilon)\left(\frac{f}{n}\right)\right\} - \left\{\theta - (\delta + \varepsilon)\left(\frac{f}{n}\right)\right\}$$

$$\pi \leq \varepsilon + 2\left\{(\delta + \varepsilon)\left(\frac{f}{n}\right)\right\}$$

But n = 3f+1, therefore

$$\pi \leq \varepsilon + 2(\delta + \varepsilon)\left(\frac{f}{3f+1}\right)$$

Hence proved.

Proposition 1. The worst possible precision of WASA for large value of f is within:

$$\pi \leq \varepsilon + \frac{2(\delta + \varepsilon)}{3}$$

Proof. From Theorem 1 we have

$$\pi \leq \varepsilon + 2(\delta + \varepsilon)\left(\frac{f}{3f+1}\right)$$

If the number of malicious clocks is very large i.e. f ≫ 1 then 3f+1 ≃ 3f hence

$$\pi \leq \varepsilon + \frac{2(\delta + \varepsilon)}{3}$$

This is the worst possible Π for large value of f.

One of our aims of designing WASA is to ensure that the good clocks never deviate from (δ + ε) as defined in Agreement Property.

Proposition 2. WASA guaranteed satisfaction of the Agreement Property.

Proof. The worst possible precision guaranteed by WASA is given by

$$\pi \leq \varepsilon + \frac{2(\delta + \varepsilon)}{3}$$

Since $\varepsilon \ll \delta$,

$$\pi \leq \varepsilon + \frac{2(\delta + \varepsilon)}{3} < (\delta + \varepsilon)$$

Hence WASA guaranteed Agreement Property.

From Proposition 2 we can establish that WASA ensures all good clocks are consistently within < (δ + ε) . This completes the proof of correctness.

The worst case precision by SWA^{det}_{mean} [Pfluegl and D. M. Blough , 1995] is given as for

n ≥ 4f

$$\pi \leq \varepsilon + f\frac{(\delta + \varepsilon)}{n - f} + f\frac{(2\delta + \varepsilon)}{n - f + 1}$$

If f ≫ 1 and taking δ ≫ ε , WASA is at least 33% tighter.

4.3.3. Analysis for optimal weight assignment for weighted average

Weight assignment enables us to mitigate the effect of bad clocks, which cannot be segregated from the minimum variant window. It can be appreciated that maximum of the bad clocks which manage to stay inside the minimum variant window are malicious clocks. For assignment of the weights, we first find out the mean, μ and standard deviation, σ, of the minimum variant window. Then clocks within one ± σ distance from μ are assigned weight ω_1, where μ is the mean and σ is the standard deviation of the window. Clocks within ± σ to ± 2 σ are assigned weight ω_2 and clocks within ± 2 σ to ± 3 σ are assigned weight ω_3. Finally, clocks beyond ± 3 σ are assigned zero weight.

The distribution of all clocks for the worst-case analysis is given in figure 2.6 below. In the minimum worst-case scenario all the malicious clocks are around the smallest valued good clock. The algorithm also makes the adjustment all good clocks that are outside the minimum variant window. The adjustment make *f* such good clocks take same value as the largest good clock inside the minimum variant window. Therefore as shown in the figure, there are *f+1* clocks concentrating on either edge of the window. A similar explanation is also true for maximum worst-case analysis. There will be *f-1* clocks left between these clocks, which are on the edge of the window.

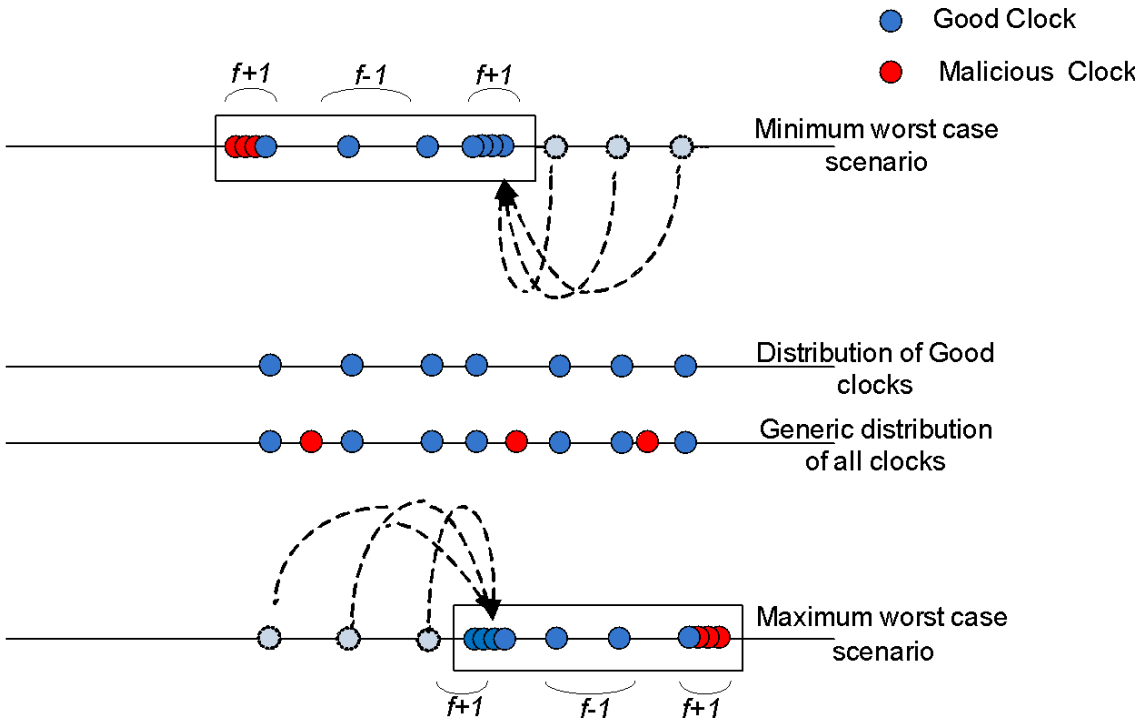

Figure 4.7. Concentration of clocks in Worst-case clock distribution

As most of the world values are normally distributed, approximately sixty eight percent of these n clock values are within ± σ of the mean as shown in figure 2.7 below. It can be easily seen that the distribution pattern of the good clocks other then which are outside the minimum variant window are altered by the algorithm. Hence, good clocks within ± σ of the mean are not altered. If we $\omega_1 = 1$, $\omega_2 \ll \omega_1$ and $\omega_3 = 0$, then all the clocks on the edges are negated for further calculations. The remaining clocks in the minimum worst-case window and in the maximum worst-case window are those, which were originally within ± σ of the original clock distribution obtained after, stage one of WASA. This shows that the means of the minimum worst-case window and maximum worst-case window are within 2σ of each other at most. The maximum deviation δ is approximately equal to 6σ as there are

99.7% of the total clocks within ± 3σ from the mean. Therefore, the means of the minimum worst-case window and maximum worst-case window is at most one-third of δ. Hence the weights, $\omega_1 = 1$, $\omega_2 \ll \omega_1$ and $\omega_3 = 0$, offer a much better precision. The optimality of these weights is not established.

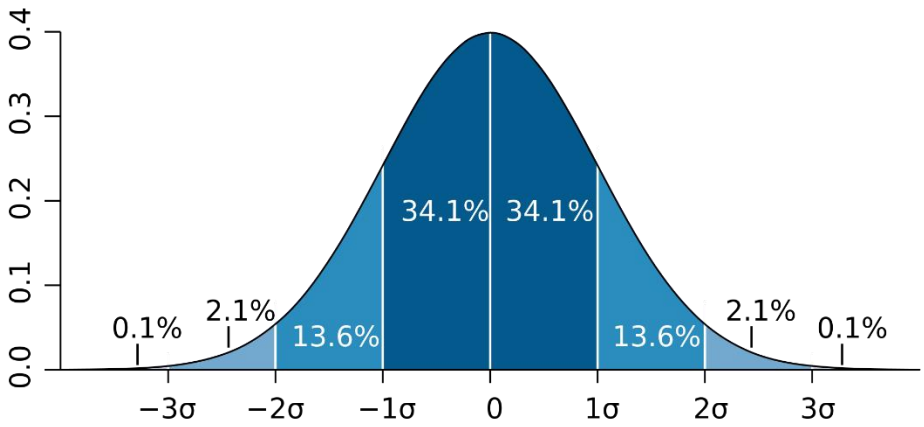

Diagram courtesy Wikipedia.org

Figure 4.8. Normal Distribution

4.3.4. Pseudo code of WASA

We present pseudo code of our algorithm in this section. The algorithm executes in rounds and the resynchronization is done at each round. Each node in each round executes the process presented in the pseudo code below. First, the variables used in the algorithm are given then the subroutines used are explained subsequently.

Global variable:

n : Number of clocks in the network

f : Number of faulty clocks in the network

w_1, w_2, w_3 : weights

Local variable:

all_node_clock : Clock of all nodes of the network

Clock_array : Array whose members are the clocks of the network

Clock_array_sort : Array whose members are sorted in an order

Var_clock_win : Return minimum value variance from an array whose members are variances

min_ Var_clock_win : Array of clock whose variance is minimum

left_ Var_clock_win : Array whose members are member of Clock_array and not members of min_ variance_clock_win but smaller than the first of min_ Var_clock_win

right_ Var_clock_win : Array whose members are members of Clock_array and not member of min_ Var_clock_win but larger than the last member of min_ Var_clock_win

Complete_clock : Final clock array whose members consists of n clocks

mean_ complete_Var_clock_win: mean of the array complete_Var_clock_win

sd_ complete_Var_clock_win: standard deviation of the complete_Var_clock_win

mn : mean_ complete_Var_clock_win

sd : std_ complete_Var_clock_win

W_1_ complete_Var_clock_win: Array whose members are within $\mp$ sd of mn

W_2_ complete_Var_clock_win: Array whose members are within $\mp$ 2sd of mn but not member of W_1_ complete_Var_clock_win

W_3_ complete_Var_clock_win: Array whose members are within $\mp$ 3sd of mn but not member of W_2_ complete_Var_clock_win

new_clock : New clock value

Subrountines:

Request_all () : Request all nodes to send their clocks

Receive() : All clock values sent is received by the requesting node

Sort() : Sort the members of an array in ordered fashion

Variance() : Calculate the variance of an array

Min() : Return the minimum element of an array

Element() : Return the number of members of an array

Cat() : Concatenate multiple arrays into a single array

Mean() : Return the mean of the members of an array

Std() : Return the standard deviation of the members of an array

Pseudo-Code:

Each node will perform this process at the start of T_r.

Request_all (all_node_clock)

Receive (all_node_clock):

Clock_array[n] = [$clock_1$, $clock_2$,...,$clock_n$];

Clock_array_sort = **sort**(Clock_array[n]);

For i=1 to f+1

Var_clock_win(i) = **Variance** (Clock_array_sort[i:(n-f+i-1)]);

i++;

end

min_ Var_clock_win=**min**(Var_clock_win);

If (**element**(min_Var_clock_win) < **element**(clock_array))

left_win(1,1:**element**(left_Var_clock_win))= min_ Var_clock_win[1,1];

right_win(1,1:**element**(left_Var_clock_win))= min_ Var_clock_win[1,n-f];

complete_Var_clock_win=**cat**(left_win, min_var_clock_win, right_win);

else

complete_Var_clock_win= min_var_clock_win;

end

mean_ complete_Var_clock_win=**mean** (complete_Var_clock_win);

mn= mean_ complete_Var_clock_win

sd_ complete_Var_clock_win = **std** (complete_Var_clock_win);

sd= sd_ complete_Var_clock_win;

For i=1 to n

If (mn + sd ≥ complete_Var_clock_win(1,i) ≥ mn-sd)

W_1_ complete_Var_clock_win(1,i)= complete_Var_clock_win (1,i)

elseif (mn + 2*sd ≥ complete_Var_clock_win(1,i) ≥ mn-2*sd)

W_2_ complete_Var_clock_win(1,i)= complete_Var_clock_win (1,i)

elseif (mn + 3*sd ≥ complete_Var_clock_win(1,i) ≥ mn-3*sd)

W_3_ complete_Var_clock_win(1,i)= complete_Var_clock_win (1,i)

else

complete_Var_clock_win(1,i)= 0

end

end

new_clock= (w_1* w_1_ complete_Var_clock_win+

w_2* w_2_complete_Var_clock_win+ w_3* w_3_ complete_Var_clock_win)/(w_1+w_2+w_3)

4.4. Message Complexity Analysis

We have designed our algorithm for a completely connected static network. Here all the nodes are connected to the other nodes with a bi-directional communication links. All these nodes communicate by passing messages. For calculation of the message complexity let us take any one node from this network. This node, n_i , has a link to every node in the network. During the course of the resynchronization process n_i request all nodes to share their clock value. All these nodes then response to this request by sending their clock value to n_i. This process take *2N* messages if the number of nodes in the network is *N*. For *N* nodes the total number of messages exchanged is $2N \times N = 2N^2$. Therefore, the message complexity of the algorithm is **O(N^2)**.

4.5. Time Complexity Analysis

Here in this section we analysis your algorithm to find out the time complexity. Time complexity gives the relationship between the computing time and the input data of the algorithm. The knowledge of time complexity of the algorithm facilitates optimization of the algorithm computing time. The time complexity is computed in terms of N, where N is the number of times a particular statement is executed. For calculation of the time complexity of the algorithm, we have followed the following:

(i) The running time of a statement in the algorithm that does not have any loop is considered constant.

(ii) The running time of a single loop is taken as linear, that is multiple of N.

(iii) The running time of a double loop is quadratic.

(iv) Lastly, the running time of an iterative statement is considered logarithmic.

The maximum running time take by any subroutine is by sort() and linear. In the algorithm other then the subroutine there are one loop and two iterative statements. The second iterative statement -

"For i=1 to n

If (mn + sd ≥ complete_Var_clock_win(1,i) ≥ mn-sd)

W_1_ complete_Var_clock_win(1,i)= complete_Var_clock_win (1,i)

elseif (mn + 2*sd ≥ complete_Var_clock_win(1,i) ≥ mn-2*sd)

W_2_ complete_Var_clock_win(1,i)= complete_Var_clock_win (1,i)

elseif (mn + 3*sd ≥ complete_Var_clock_win(1,i) ≥ mn-3*sd)

W_3_ complete_Var_clock_win(1,i)= complete_Var_clock_win (1,i)

Else

complete_Var_clock_win(1,i)= 0

end

end"

takes the maximum running time. The statement requires **N*log(N)** since there is a loop and an iterative statement combine. Hence the time complexity of the algorithm is **O(N*log(N)).**