# Improvements to a Protocol for the Maintenance of Common Data in Distributed Systems

Dai Tho Nguyen
VNU University of Engineering and Technology
Hanoi, Vietnam
nguyendaitho@vnu.edu.vn

Ho Thuan
Institute of Information Technology, VAST
Hanoi, Vietnam
hothuan@vast.ac.vn

Thanh Le Dinh
VNU University of Engineering and Technology
Hanoi, Vietnam
thanhld@vnu.edu.vn

***Abstract*—In this paper, we suggest some modifications to a protocol proposed by Awerbuch and Schulman in [3] for the maintenance of common data in distributed systems so that both its time and communication complexities are reduced. The first change that makes Awerbuch-Schulman protocol better is to eliminate unnecessary messages used in the original version. The second change that also improves the protocol is to reduce the size of process messages. In addition, we suggest a self-stabilizing version of the Awerbuch-Schulman protocol as well.**

***Keywords-broadcast with partial knowledge, process, incremental update, discrepancy, self-stabilizing.***

## I. INTRODUCTION

Common objects in a distributed system are subject to occasional changes. After changes, it is necessary for each site to update its view so that the view reflects the current state of the common objects. This problem is known as the maintenance of common data in distributed systems and has been studied for decades. Among protocols for solving this problem [1-3], the Awerbuch-Schulman protocol [3] is the best one since it has poly-logarithmic overhead in both time and communication complexities. Other protocols require polynomial overhead in at least one of these measures. Moreover, as far as we know, up to now, there is no self-stabilizing protocol to this problem. These observations have motivated us to do the current work.

The main reason we are interested in the Awerbuch-Schulman protocol is that the techniques used in this paper can be used to improve the time complexity of the Awerbuch-Cidon-Kutten protocol for the communication-optimal maintenance of a spanning tree as indicated by the authors themselves [1]. The construction and maintenance of a spanning tree play an essential role in the design of many distributed algorithms, including broadcast, multicast, reset, routing, termination detection, etc. Relying on the Awerbuch-Cidon-Kutten, we have proposed an efficient and message-optimal multicast routing protocol in mobile ad-hoc networks [5]. We hope that our routing protocol can be improved in terms of time complexity.

In this paper, we suggest some modifications to the Awerbuch-Schulman protocol so that its overheads in both time and communications are reduced. The improvements consist of the elimination of unnecessary messages used in the original version and the reduction of the size of process messages. In addition, we suggest a self-stabilizing version of the protocol as well.

We use terms already introduced in [3] when explaining our improvements. For details of the problem definition and Awerbuch-Schulman protocol, please refer to [3].

## II. OUR IMPROVEMENTS TO THE AWERBUCH-SCHULMAN PROTOCOL

### A. *Observation*

The Awerbuch-Schulman protocol does reconcile time with communication requirements. By rough estimate, i.e. in word model, the communication complexity of the protocol is $O(n+\Delta)$, where $n$ is the number of sites other than the source and $\Delta$ is the total number of incorrect bits. Each correction message has the size of $(1 + \log m)$ bits, in which the first bit has the value of 1 indicating that message is a correction one, $\log m$ bits left present the height of incorrect bit, where $m$ is the size in bits of common data. Each process message has $3(1+\log m)$ bits, in which the first bit has the value of 0 indicating that message is a process one, $\log m$ bits present the altitude of the sender process, $\log m$ bits present the height of the sender process, $\log m$ bits indicate the number of incorrect bits that the sender process has corrected, one bit indicates whether the sender process is in open or split mode, and the last bit indicates whether the sender process is the main process. According to Awerbuch and Schulman's analysis, there are exactly $\Delta$ correction messages and at most $n+\Delta$ process messages used in each execution of their protocol, thus there are at most $\Delta*(1 + \log m) + (n+\Delta)*3(1 + \log m) = (3n+4\Delta) \log m + 3n + 4\Delta$ bits of messages used in each execution. Obviously, if $n$ or $\Delta$ is quite large, the number of bits of messages is much greater than $(n+\Delta) \log m$. In fact, we can reduce the number of messages and the number of bits of messages used in the Awerbuch-Schulman protocol, so does the time of each execution (still remain the O(*) complexities of the original protocol).

Observe that, in the Awerbuch-Schulman protocol, when a process *Q* enters in split mode, it generates two child processes: the upper and the lower ones; the children “go ahead” and do its parent’s duty of cleaning a subarray, whereas *Q* “tags behind” its children and do nothing except “relaxing”; until two child processes have terminated (at a same site), *Q* finishes its “relaxing” period and works again or terminates. By whatever *Q* does when its children terminate, the “tagging behind” period of *Q* is waste. This is the point we use to make our improvement in the Awerbuch -Schulman protocol.

## *B. Improved Issues*

The idea behind our improvement is simple. When entering in split mode, *Q* generates two children then terminates rather than tags behind its children. When two *Q*’s child processes terminate, *Q* will be “revived” if its duty hasn’t been completed, i.e. in cases where current site is not an $m_Q$-column one or *Q* is the main process. The revivification of *Q* is taken by its upper child. When *Q*’s upper child, <$y_Q$ , $m_Q/2$, ., ., .>, reaches its destination at an $m_Q/2$-column, if this is an $m_Q$-column, it knows that *Q* needs to terminate, so it doesn’t revive *Q* (unless *Q* is the main process); otherwise it knows *Q* needs to be revived to continues *Q*’s duty, so it sends a process message <$y_Q$ , $m_Q$, *0*, ., .> to the site’s successor before its termination, thus *Q* is revived. The reappearance of *Q* in this manner is equivalent with *Q*’s returning to open mode when two its child processes terminate as in the original version of the Awerbuch-Schulman protocol.

The key issue is how a process knows itself that it is an upper child process (of another process)? In process messages, in the form of <*y, h, e, b, c*>, *b* is no longer necessary because all processes are in open mode (rather than entering in split mode and tagging behind its children, *Q* terminates itself, then it will be revived). An easy way to distinguish upper child processes from others is to use *b*: *b* = 1 if process is an upper child (of another process), *b* = 0 otherwise. However, this method leads to a recursive problem that cannot be solved. That is, when reviving parent process, how does a child process know that its parent is also an upper child process (of another process), and similarly, how does parent process know that the grandparent process is also an upper child process (of another process), etc.

Fortunately, both these two issues can be solved smartly and effectively. A process is identified as an upper child (of another process) if the quotient of the altitude of the process plus 1 with the height of the process is an even number, i.e. $(y+1)/h = 2i, i = 1, 2, ...$

Moreover, *c* component in a process messages is unnecessary since we can identify the main process as the unique process with the altitude of *m*-1 and the height of *m*. An upper child of the main process is a process with the altitude of *m*-1 and the height of *m*/2, therefore, before this process terminates, it revives the main process if its site isn’t the last one.

Thus, a process message can be reduced in size to the form of <*y, h, e*> - 2 bits smaller than process message in the original version of the Awerbuch-Schulman protocol. The time for a process message to move from one site to that site’s successor is decreased by 2 time units.

The revivification of a process is accomplished as follows. Process *Q*, <$y_Q$, $m_Q$, $e_Q$>, before terminating at an $m_Q$-column that isn’t the last column, checks itself if it is an upper child (of another process) by checking whether $(y_Q+1)/m_Q$ is an even number. If this is true, then it checks whether its parent is the main process ($y_Q = m-1$ and $m_Q = m/2$) or $m_Q$-column site isn’t an $2*m_Q$-column, *Q* will send a process message <$y_Q$, $2*m_Q$, *0*> to its site’s successor. Process message <$y_Q$, $2*m_Q$, *0*> does revive *Q*’s parent.

Our improved Awerbuch-Schulman protocol is described in detail in Appendix 1.

In this protocol, the source simply generates main process by calling *Process*(*m*-1, *m*, 0); other processes will correct incorrect bit upon receiving correction message *Error*(*j*), and will generates next process by calling *Process*(*y*, *h*, *e*) upon receiving process message *MoveForward*(*y*, *h*, *e*).

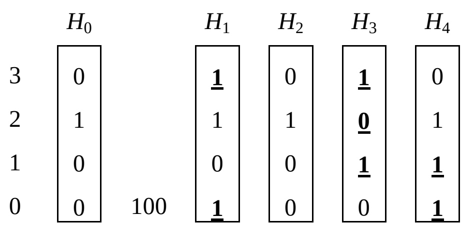
The execution of a process at a site is implemented by procedure *Process*(*y*, *h*, *e*). After completing the duty of cleaning a segment of bits in its site’s successor, if it isn’t at an *h*/2-column, a process will move to its site’s successor. Otherwise, i.e. it’s at an *h*-column, if the process is the main process, it starts cleaning next *m*-rectangle; else if the process is an upper child (of another process), it revives its parent before its termination; else the process does nothing more but terminate. If a process is at an *h*/2-column that isn’t an *h*-column, depending on the value of *e*, it enters in split mode or simply resets *e* to 0 and starts a new open mode.

## *C. Example*

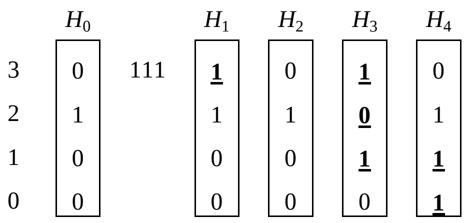
An execution of the improved Awerbuch-Schulman protocol is given in Figure 1. In this example, the chain of hosts is $H_0$-$H_1$-$H_2$-$H_3$- $H_4$. $H_0$ has a 4-bit data which may change occationally. Each of $H_1$, $H_2$, $H_3$, $H_4$ maintains a copy of the 4-bit data from $H_0$. The execution consists of a series of configurations. Each configuration is presented as the status (data) of hosts and actions of processes. Four bits of data of a host is presented in a box. The incorrect bits are displayed in underlined and bold style. That a process performs an action in a host is presented in the form *host: process-id<y, h, e> action*. For example, $H_0$: #main<3, 4, 2> sends(111) means the process <3, 4, 2> is given #main as its identifier, is running on $H_0$, sends a message whose content is 111 to the next host ($H_1$).

|  | $H_0$ | $H_1$ | $H_2$ | $H_3$ | $H_4$ |
|---|---|---|---|---|---|
| 3 | 0 | $\underline{\mathbf{1}}$ | 0 | $\underline{\mathbf{1}}$ | 0 |
| 2 | 1 | 1 | 1 | $\underline{\mathbf{0}}$ | 1 |
| 1 | 0 | 0 | 0 | $\underline{\mathbf{1}}$ | $\underline{\mathbf{1}}$ |
| 0 | 0 | $\underline{\mathbf{1}}$ | 0 | 0 | $\underline{\mathbf{1}}$ |

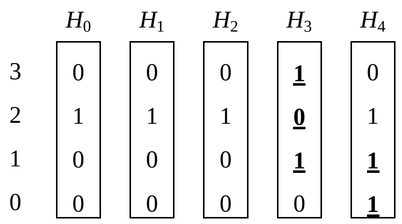
$H_0$: #main<3, 4, 0> created



$H_0$: #main<3, 4, 1> sends(100)



$H_0$: #main<3, 4, 2> sends(111)

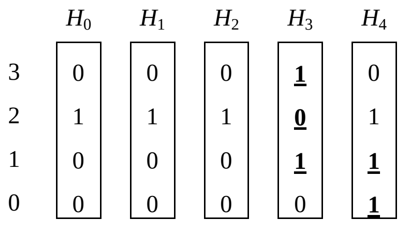


$H_1$: #main<3, 4, 2> arrives

$H_1$: #uchild<3, 2, 0> created

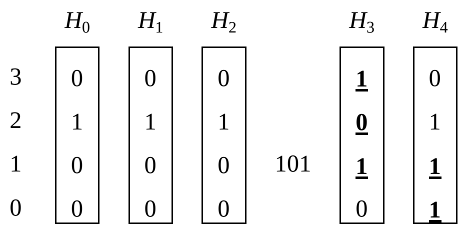
$H_1$: #lchild<1, 2, 0> created

$H_1$: #main<3, 4, 1> terminates



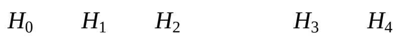
$H_1$: #uchild<3, 2, 0> waits

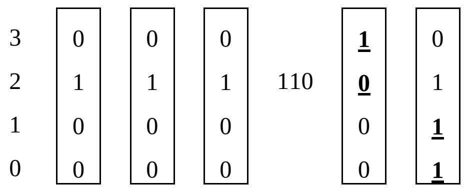
$H_2$: #lchild<1, 2, 0> arrives



$H_2$: #uchild<3, 2, 0> arrives

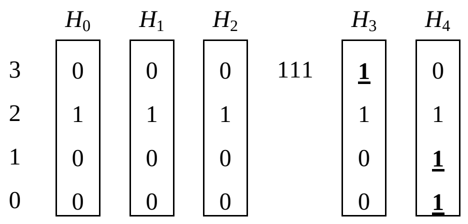
$H_2$: #lchild<1, 2, 1> sends(101)





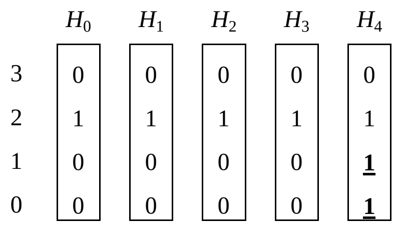
$H_2$: #uchild<3, 2, 1> sends(110)

$H_3$: #lchild<1, 2, 1> arrives

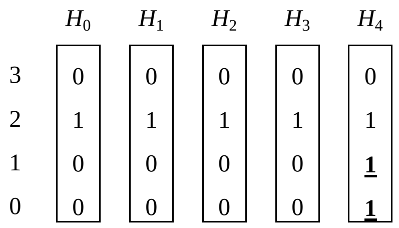


$H_2$: #uchild<3, 2, 2> sends(111)

$H_3$: #lchild<1, 2, 1> terminates



$H_3$: #uchild<3, 2, 2> arrives



$H_3$: #main<3, 4, 0> revived

$H_3$: #uchild<3, 2, 2> terminates

**Figure 1.** *An execution of the improved Awerbuch-Schulman protocol.*

## III. Correctness and Complexities

As with the original Awerbuch-Schulman protocol, it's easy to see the correction of our improved protocol since in each execution, each incorrect bit has exactly one correction message, thus is reversed once, and all correct bits are cleaned but aren't reversed.

With our improvement, not only does the number of process messages decrease but remaining process messages are also smaller in size. Therefore, overheads in time and communications are reduced.

To estimate the effectiveness of our improvement, first of all, we have Lemma 1 about the Awerbuch-Schulman protocol.

**Lemma 1.** *In each execution of Awerbuch-Schulman protocol, for each site $P_i$, the number of processes that run on $P_i$ is an odd number $p_i = 2k_i+1$, $k_i \geq 0$, in which $k_i$ processes are in split mode and $k_i+1$ processes are in open mode.*

For proof of Lemma 1, see Appendix 2.

**Assertion 1.** *Let T be the number of process messages used in an execution of the Awerbuch-Schulman protocol. In the same execution, our improved protocol saves (T-n)/2 process messages.*

For proof of Assertion 1, see Appendix 3.

## IV. Self-stabilizing Version

A self-stabilizing system is a system that can eventually exhibit legitimate behavior regardless of its initial state [4]. In other word, starting at an arbitrary state a self-stabilizing system ensures that it will reach consecutive legal states.

A self-stabilizing system can tolerate any (type and amount of) fault. Occurrence of faults brings system to arbitrary state. But if after the burst of faults there is a long enough period during which no fault occurs then system will reach legal states by its self-stabilizing nature.

With our improved Awerbuch-Schulman protocol, a configuration is legitimate if in this configuration each site has its own data as the same as that of the source. An execution is legitimate if every configuration in it is legitimate.

In order to have self-stabilization property in our improved Awerbuch-Schulman protocol, some issues need to be solved. First, the source needs to periodically start protocol runs, and the others involve in these runs. Second, neighbor-knowledge assumption needs to be held. Assume that in initial state $P_1$'s data is totally different from $P_0$'s knowledge about it (each bit of $P_1$'s data differs from the corresponding bit of $P_0$'s knowledge about $P_1$'s data). Processes on $P_0$ will send correction messages for correct bits of $P_1$'s data but incorrect bits. Therefore, $P_1$'s data will always be different from that of the source. This problem may occur in other sites, too. Thus, if the neighbor-knowledge assumption isn't held then protocols will not be self-stabilizing. This observation suggests that to gain self-stabilization in these protocols we must hold neighbor-knowledge assumption. In order to do so, each site except the source needs to send periodically its own data in a knowledge message to previous site in order to update previous site's knowledge.

Our self-stabilizing improved Awerbuch-Schulman protocol is described in detail in Appendix 4. Assume that faults may occur in the period from 0 to $T$ but there is a long enough period follow $T$ during which no fault occurs. Let $C_T$ be the system configuration at time $T$. O($m$) time units follow $T$, all sites will complete sending knowledge message to previous site as well as receiving another knowledge message from next site, thus each site has its correct knowledge about its following site. One period after that, all sites will have its data as the same as that of the source. Time complexity (of a convergence) of our self-stabilizing improved Awerbuch-Schulman protocol is as the same as that of our non-self-stabilizing one. Message complexity (of a convergence) of our self-stabilizing improved Awerbuch-Schulman protocol is greater than that of our non-self-stabilizing one since at least $n*m$ bits of knowledge messages are used in each convergence.

## V. Conclusion

Our improvements to the Awerbuch-Schulman protocol saves time and messages for it and make it self-stabilizing. Because of the importance of the maintenance of common data in distributed systems problem, one continuously finds effective, stable protocols to this problem. Our future work is to find such another protocol.

## References

[1] Baruch Awerbuch, Israel Cidon, and Shay Kutten, "Optimal maintenance of a spanning tree," Journal of the ACM, 55(4), Sep. 2008.

[2] Baruch Awerbuch, Israel Cidon, Shay Kutten, Yishay Mansour, and David Peleg, "Broadcast with partial knowledge," In Proc. 10th ACM Symp. on Principles of Distributed Computing, 1991.

[3] Baruch Awerbuch, and Leonard J. Schulman, "The maintenance of common data in a distributed system," Journal of the ACM, 44(1): 86-103, Jan. 1997.

[4] Edsger W. Dijkstra, "Self-stabilizing systems in spite of distributed control," Comm. ACM 17: 643-644, November 1974.

[5] Hai Trung Nguyen and Dai Tho Nguyen, "An efficient and message-optimal multicast routing protocol in mobile ad-hoc networks," In Proc. of the Int. Conf. on Advanced Technologies for Communications, 2011.

## Appendices

**Appendix 1.** *Our improved Awerbuch-Schulman protocol*

```
The source:
    Process(m-1, m, 0) // Generate the main process

Other sites:
    Upon receiving Error(j) message from previous site
        Correct (reverse) bit with index j in my data
    Upon receiving MoveForward(y, h, e) message from previous site
        If the site is not the last one then Process(y, h, e)

Procedure Process(y, h, e)
    For each bit b_j, y-h ≤ j ≤ y-1, in next site's data
        If b_j is incorrect then
        Send **Error**(j) message to next site
        $e := e + 1$
  If the current site is an h-column then
     If I'm the main process, i.e. $y = m$-1 and $h = m$, then
        Start cleaning next m-rectangle by sending **MoveForward**($m$-1, $m$, 0) message to next site.
     Else if I'm an upper child, i.e.($y$+1)/$h$ is an even number, then
        Revive parent process by sending **MoveForward**($y$, 2*$h$, 0) messages to next site.
  Else if current site is an $h$/2-column then
     If $e > h/2$ then enter in split mode by sending **MoveForward**($y$-$h$/2+1, $h$/2, 0) then **MoveForward**($y$, $h$/2, 0) messages to next site.
     Else starts a new open mode in next h-rectangle by sending **MoveForward**($y$, $h$, 0) message to next site.
  Else, move to next site by sending **MoveForward**($y$, $h$, $e$) to next site.
```

**Appendix 2.** *Proof of Lemma 1*

In each execution of the Awerbuch-Schulman protocol, parent processes tag behind its children, thus if $p$ runs on $P_i$ then all $p$'s ancestors will run on $P_i$. Therefore, if we regard each process that runs on $P_i$ as a node, and each parent-and-child relation between two processes that run on $P_i$ as a link, then all processes that run on $P_i$, along with parent-and-child relations between them, form a binary tree $T_i$ (each parent process have exactly two children) with the main process at the root of $T_i$.

In addition, in each execution of the Awerbuch-Schulman protocol, if $p$ isn't the main process that run on $P_i$ then its twin sibling, $q$, runs on $P_i$ , also, because two twin processes have the same path (are given birth at the same site and terminate at another same site). Thus, $T_i$ is a complete binary tree, has $p_i = 2k_i + 1$ nodes, $k_i \geq 0$, in which $k_i$ internal nodes and $k_i+1$ leaves. Note that each internal node is a parent process, therefore, in split mode; and each leaf is a process that hasn't split, therefore, in open mode. □

**Appendix 3.** *Proof of Assertion 1*

From the Lemma 1, it is easy to see that the number of split-mode process messages is ($T$-$n$)/2. These split-mode process messages are eliminated in our improved protocol.□

**Appendix 4.** *Our self-stabilizing improved Awerbuch-Schulman protocol*

```
**The source:**
  Do periodically:
     **Process**($m$-1, $m$, 0) //Generate the main process
     Upon receiving knowledge message from next site:
        Update the view about next site's data
**Other sites:**
  Upon receiving **Error**(j) message from previous site
  Correct (reverse) bit with index j in my data
  Upon receiving **MoveForward**($y$, $h$, $e$) message from previous site
     If not the last site then **Process**($y$, $h$, $e$)
  Upon receiving knowledge message from next site:
     Update the view about next site's data
  Do periodically:
     Send knowledge message to previous site.

*Procedure* **Process**($y$, $h$, $e$)
  For each bit $b_j$, $y-h \leq j \leq y-1$, in next site's data
     If $b_j$ is incorrect then
           Send **Error**(j) message to next site
           $e := e + 1$
  If current site is an h-column then
     If I'm the main process ($y = m$-1 và $h = m$) then
        Start cleaning next m-rectangle by sending **MoveForward**($m$-1, $m$, 0) message to next site.
     Else if I'm an upper child (($y$+1)/$h$ is an even number) then
        Revive parent process by sending **MoveForward**($y$, 2*$h$, 0) to next site.
  Else if current site is an $h$/2-column then
     If $e > h/2$, enter split mode by sending **MoveForward**($y$-$h$/2+1, $h$/2, 0) then **MoveForward**($y$, $h$/2, 0) messages to next site.
     Else starts a new open mode in next h-rectangle by sending **MoveForward**($y$, $h$, 0) message to next site.
  Else, move to next site by sending **MoveForward**($y$, $h$, $e$) to next site.
```