FRI-ONLINE-1-CCT1-02

DISTRIBUTED RING-BASED MUTUAL EXCLUSION WITH GRACEFUL DEGRADATION[2]

Assoc. Prof. Milen Loukantchevsky, PhD, IEEE & ACM Member
Department of Computer Systems & Technologies,
University of Ruse "Angel Kanchev"
phone: 0877 303 850
e-mail: mil@ieee.org

Abstract: *Under "mutual exclusion" is understood preventing of any opportunity more than one active object (process, thread, task) to access a shared resource at a time. The distributed ring-based (aka token-ring) mutual exclusion algorithm is executed over logical circular topology. Mainly due the chosen topology the ring-based algorithm is the simplest of this kind. Its pros are simplicity and minimalistic preliminary information required to be known a priori from each system process. The main drawback of this attractive algorithm in its basic definition is the strong presumption of absolute system reliability which makes it impractical. After all, the failure model of distributed systems itself assumes that failures should not be treated as exceptions but as a norm.*

In a previous work is described a fault-tolerant version of the classical distributed ring-based mutual exclusion algorithm without communication ring reconfiguration (Scheme 1). Here is described a modified version of that algorithm (Scheme 2) with special kind of ring reconfiguration - graceful degradation. Both recovery schemes guarantee failure recovery from any kind of multiple faults, and thus eliminate presumption of full system reliability. In the Scheme 1 the recovery of a faulty process is awaited to recover full system availability. With here proposed Scheme 2 the system recovery begins as soon as a process failure is detected. This is at the expense of excluding of the faulty process from the system configuration. That leads to shrinking of the ring (one-way resiliency) and represents a kind of graceful degradation. Compared to the first scheme, the second is more operational as it eliminates the delay needed to repair the faulty process. In addition, the next recovery Scheme 3 means to stretching of the ring for two-way resiliency.

Keywords: *Distributed Systems, Fault-tolerance, Failure Recovery, Mutual Exclusion, Resiliency, Token-Ring*
ASJC Codes: *1701, 1712*

INTRODUCTION

The mutual exclusion (*ME*) was first introduced by Edsger Dijkstra thus marking the computer science of concurrency [9]. The problem is to prevent any opportunity more than one *active object* (process, task, thread, etc.) to access a shared resource at a time. While in multitasking operating systems this is ensured by global system variables under kernel control (known usually as *kernel objects*), in distributed systems the solution is based on the idea of the *critical section*. Where the *critical section* is defined a section of the code of the active object in which the shared resource is accessed [1, 3, 5, 6, 11, 13, 14]. The difference is imposed by the different underlying computational models – global vs distributed memory model [3, 4, 12]. The main consequence in the second case is the need to use *message passing* over communication channels as the only means of inter process interaction.

There are three requirements for distributed mutual exclusion algorithms: *safety* (*ME1*), *liveness* (*ME2*) and *fairness* (*ME3*). Where the first two are mandatory, while the latter one is optional.

Distributed mutual exclusion algorithms are split into two big families: *permission based* and *token-based* [13]. One of the most popular token-based algorithms is the *circular* aka *ring-based* or *token-ring* one. It supposes homogeneous system from n identical processes $P_1, P_2, \dots, P_n$ connected in a logical circular topology. The canonical scheme of a such *ring-based* distributed system consisted for example of $n = 6$ processes is shown at Fig. 1.

There is a single *ME* token, a service message that moves in the communication ring clockwise. As only the process possessing the *ME* token can be in the critical section, the safety requirement *ME1* is satisfied. The liveness requirement *ME2* is satisfied by the circular topology itself, since the

[2] Докладът е представен на заседание на секция 3.2 на 29 октомври 2021 с оригинално заглавие DISTRIBUTED RING-BASED MUTUAL EXCLUSION WITH GRACEFUL DEGRADATION

ME token eventually will arrive into every one process over the ring. The *ME* token is generated once by the process with the largest identifier, called *coordinator* (the P_6 in our example). The coordinator is determined during the system initialization by a *distributed election algorithm* [3, 5, 6]. Should be noted that the ordering of the processes in the ring is irrelevant (at Fig. 1 they are arranged consecutively for convenience only).

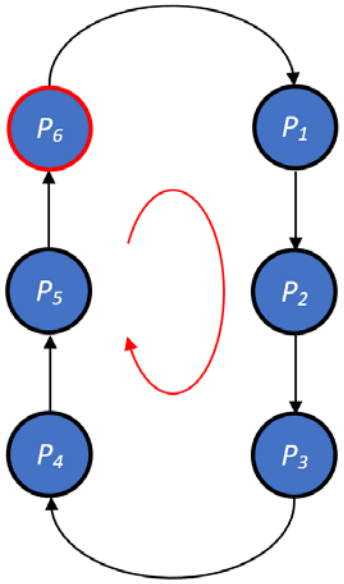

Fig. 1. The *canonical scheme* of a ring-based distributed system ($n = 6$).

A major disadvantage of this basic variant of the *ring-based* distributed mutual exclusion algorithm is the requirement of absolute system reliablity - neither processes nor channels could crash.

That is the reason to formulate our *objective*: through suitable modification to overcome the main shortcoming of the basic algorithm - the inadmissibility of process failures.

EXPOSITION

1. Failure Recovery Schemes

When talk about *failure recovery* we should first clarify the *process failure* as a failure of the process P_i itself, a failure of the section of the input channel of the P_i or a failure of the section of the output channel for which P_i is responsible for (Fig. 2). The simplest failure class, supposed here, is *fail-stop*.

Fig. 2. Graphical interpretation of the *process failure*

Three working *recovery schemes* are considered through the overall project "*Class of Fault-Tolerant Distributed Algorithms for Mutual Exclusion over Elastic Logical Ring Topology*". They evolve consistently and complement each other:

- *Scheme 1* (Failure Recovery Without Reconfiguration)
- *Scheme 2* (Failure Recovery with One-way Reconfiguration)
- *Scheme 3* (Failure Recovery with Two-way Reconfiguration)

Under *Scheme 1* is assumed that no changes in the system configuration are made. In case of a process failure the hole system stops (at least in respect to *ME* algorithm). Only after the faulty process is restored, all other processes resume normal operation from the state they were in at the time of the failure and thereafter the *ME* token is restored. In doing so, strict compliance with the *ME1* and *ME2* requirements is ensured. This modified distributed ring-based mutual exclusion algorithm designated as *Mx1ME* allows distributed system recovery from multiple faults [7].

With *Scheme 2* the system recovery begins as soon as a process failure is detected. This is at the expense of excluding of the faulty process from the system configuration. That leads to *ring shrinking* (*one-way resiliency*) and represents a kind of *graceful degradation*. Compared to the first

scheme, the second one is more operational as eliminates the time to repair of the faulty process. The whole system resumes its normal operation with *latency* determined by the next times: the time *to detect* the process failure, the time *to isolate* of the faulty process, the time *to recover* of the *ME* marker.

As an example, at Fig. 3 is shown the *shrinking* of the initial ring configuration (Fig. 1) as result of consecutive or even simultaneous failures of processes P_2, P_4 and P_5. This recovery scheme guarantees recovery from any *fail-stop* process failures until an acceptable system degradation to minimum of k faultless processes, where $1 < k \leq n$. Such a system is *fault-tolerant* or (*n-k*) *resilient*, where (*n-k*) is the number of failures overcome. The maximum possible *one-way resiliency* is (*n-1*) when the ring collapses into one (!) point.

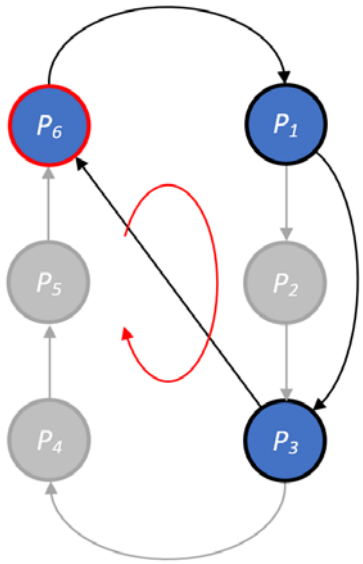

Fig. 3. Reconfiguration with *graceful degradation*

Scheme 3 supposes both exclusion of the faulty processes and inclusion (*injection*) of faultless spare processes (despite if this is a recovered or a "fresh" one process) as replacement of faulty ones. Apparently, it is a kind of *ring stretching*. So, combining both *Scheme 2* and *Scheme 3* the communication ring acquires the property of *elasticity* or *two-way resiliency*.

2. Chain of Algorithms for Resilient Ring-Based Distributed Mutual Exclusion

The first of the above series of recovery schemes is *Scheme 1*. Its restriction to recovery without ring reconfiguration allows to solve the fundamental tasks: the detection of a process failure and the *ME* token management [7]. Respectively modified distributed mutual exclusion algorithm, namely *Mx1ME*, is last in the chain of helper distributed algorithms:

- Output Channel Error Handling Algorithm, *CEH*;
- Communication Ring Checkup Algorithm, *RUP*;
- Distributed Ring-based Election Algorithm, *E*;
- Token Management and Recovery Algorithm, *MrkME*;
- Distributed Mutual Exclusion with Failure Recovery, *Mx1ME*.

The *CEH* algorithm executed by the process P_i is responsible for detection of its immediate neighbor failure while *MrkME* algorithm is responsible for the *ME* token management and recovery. It ensures that there is one and only one active *ME* token in the ring, according to strong *ME1* and *ME2* requirements. In other words, it resolves not only the *lost token* problem [1] but also the *no-duplicated token* problem. Hence this algorithm is crucial for recovery from failures. The full formalized specs of the algorithms included in the above chain is available in *__SPECS__\1.Scheme 1* subfolder of the appropriate *GitHub* repository [8]. Please note that the pseudo-code of *MrkME::OnRelease* event handler presented in [8] is changed to

OnRelease:

```
If state = HELD
    state := RELEASED
    If MrkME::strClrPending != NULL
        MrkME::OnClear()
        MrkME::strClrPending := NULL
    Else
        Send <mrk_me, Tj>
    End If
End If
```

The above rectification is necessary for the correct recovery in case of the coordinator failure in *HELD* state.

The next step forward is *one-way* reconfiguration *Scheme 2* achieved through modification of the *CEH* and the *RUP* algorithms into the *CEH.GD* and the *RUP.GD*, respectively. Thus, we get the next chain of helper distributed algorithms under *Scheme 2*:

- Output Channel Error Handling Algorithm, *CEH.GD*;
- Communication Ring Checkup Algorithm, *RUP.GD*;
- Distributed Ring-based Election Algorithm, *E*;
- Token Management and Recovery Algorithm, *MrkME*;
- Distributed Mutual Exclusion with Failure Recovery, *Mx1ME.GD*.

Where, in fact, the *Mx1ME.GD* equals *Mx1ME*. The difference in the names only reflects the difference in first two helper distributed algorithms use.

3. Output Channel Error Handling Algorithm *CEH.GD*

As said above, the *CEH* algorithm executed by the process P_i is responsible for detection of its immediate neighbor failures and for reconnection with this neighbor. The *error detection* phase is based on the wide used timeout mechanism [2] with assumption for *synchronous* distributed system [3, 5]. This phase is in general identical in both *Scheme 1* and *Scheme 2*. The difference between them is localized in the *reconnection* phase. At the *Scheme 1*, if the limit *MAX_CEH_ERR* of reconnection attempts is exhausted, the recovery failed. While at the *Scheme 2* this leads to exclusion of the current neighbor as faulty one and initiation of new series of reconnection attempts with the process after the faulty neighbor, considering him as a new likely neighbor. So redesigned algorithm *CEH.GD* requires additional information – the list of identifiers of all processes along the ring. The appropriate internal variable *ListPIds* (ALGORITHM 1.1) could be set statically at system startup or dynamically during ring checkup algorithm *RUP.GD* (ALGORITHM 1.2). This is controlled by the boolean *AutoList*.

For the example of Fig. 1 we have

$AutoList_1 = \{P_2, P_3, \dots, P_6, P_1\}$,
$AutoList_2 = \{P_3, P_4, \dots, P_1, P_2\}$,
$AutoList_3 = \{P_4, P_5, \dots, P_2, P_3\}$,
…
$AutoList_6 = \{P_1, P_2, \dots, P_5, P_6\}$,

The generalized formal specification of *CEH.GD* algorithm consists of a declarative part (ALGORITHM 1.1) and a definition of event handlers (ALGORITHM 1.2).

ALGORITHM 1.1: Declarative Part of *Pi::CEH.GD.Auto*

```
{SYSTEM CONSTANTS}
Int MAX_CEH_PERIOD      // period between reconnection attempts
Int MAX_CEH_ERR         // maximum reconnection attempts
Int MIN_K               // min number of faultless processes (max degradation), 1 ≤ MIN_K
PId i                   // process Pi identifier
PId j                   // default neighbor process Pj
Bool AutoList           // <true> if ListPIds is to be filled during RUP.GD.Auto
                        // <false> if ListPIds is known in advance (RUP used)
{SET OF STATES}
<State> := {INIT, CLOSED, OPENED, FAULTY}
```

```
{INTERNAL STATE SPACE}
State state                 // current process CEH state
RUP::state                  // current process RUP state
Int ErrorCounter            // error counter
Timer TimerCEH              // timer
List ListPIds               // list of process identifiers
PId PIdNext                 // current neighbor process identifier
```

The definitive part of the *CEH.GD* (ALGORITHM 1.2) of course should be consistent with the reactive character of the distributed systems – the system is in a stable state until some predefined event brought it to another stable state. That is why this part consists of the handlers of all events caught.

ALGORITHM 1.2: Event Handlers of *Pi::CEH.GD.Auto*

```
OnInit:
    state := INIT
    ErrorCounter := 0
    TimerCEH.Interval := MAX_CEH_PERIOD
    ListPIds.Set()
    If AutoList = true
        ListPIds.Clear()
        PIdNext := j
    Else
        ListPIds.Set()
        PIdNext := ListPIds.PopFront()
    End If

OnShow:
    ErrorCounter := 0
    ChannelOut.Open()

OnOutputConnect:
    state := OPENED
    ErrorCounter := 0
    {Start Ring Check Up Algorithm}

OnOutputDisconnect:
    state := CLOSED
    ErrorCounter := 0
    TimerCEH.Start()

OnOutputError:
    state := CLOSED
    ErrorCounter := ErrorCounter + 1
    If ErrorCounter < MAX_CEH_ERR
        TimerCEH.Start()
    Else
        If ((AutoList = false) ∪ (RUP::state = UP)) ∩ (ListPids.Size() ≥ MIN_K)
            PIdNext := ListPIds.PopFront()
            ErrorCounter := 0
            TimerCEH.Start()
        Else
            {UNRECOVERABLE FAILURE}
        End If
    End If

OnTimer:
    TimerCEH.Stop()
    ChannelOut.Open()
```

4. Communication Ring Checkup Algorithm *RUP.GD*

The *RUP* algorithm is used for communication ring integrity checkup. It starts just after the connection with the immediate neighbor is established. In turn, when it is over, starts the next link of the chain – the distributed ring-based election algorithm *E*.

The generalized formal specification of *RUP.GD* algorithm consists as well of declarative part (ALGORITHM 2.1) and definition of event handlers (ALGORITHM 2.2).

ALGORITHM 2.1: Declarative Part of *Pi::RUP.GD.Auto*

```
{SYSTEM CONSTANTS}
Int MAX_RUP_PERIOD          // period to next check
String MRK_RUP              // message type „RUP Token“
String MRK_RUP2             // message type „RUP AutoList Token“
PId i                       // process Pi identifier
PId j                       // default neighbor process Pj
CEH::AutoList               // <true> if ListPIds is to be filled during RUP
                            // <false> if ListPIds is known in advance
{MESSAGES}
<mrk_rup, i>
<mrk_rup2, i, list>

{SET OF STATES}
<State> := {INIT, DOWN, UP}

{INTERNAL STATE SPACE}
State state                 // current process RUP state
Timer TimerRUP              // timer
CEH::ListPIds               // list of process identifiers
CEH::PIdNext                // current neighbor process identifier
```

The definitive part of the *RUP.GD* (ALGORITHM 2.2) is consistent with the reactive character of the distributed systems too.

ALGORITHM 2.2: Event Handlers of *Pi::RUP.GD.Auto*

```
OnInit:
    state := INIT
    TimerRUP.Interval := MAX_RUP_PERIOD

OnOutputConnect:
    {Ring Check Up First Attempt}
    state := DOWN
    If CEH::AutoList = true
        Send <mrk_rup2, i, list.Clear()>
    Else
        Send <mrk_rup, i>
    End If
    TimerRUP.Start()

OnOutputDisconnect:
    state := DOWN

OnOutputError:
    state := DOWN

OnReceiptOf <mrk_rup, j> ∪ OnReceiptOf <mrk_rup2, j, list>:
    If j = i
        TimerRUP.Stop()
        state := UP
        list.Add(i)
        ListPIds := list
```

```
        {Distributed Election Entry Point}
        E::OnStartElection()
    Else
        If CEH::AutoList = true
            Send <mrk_rup2, j, list.Add(i)>
        Else
            Send <mrk_rup, j>
        EndIf
    End If

OnTimer:
    {Ring Check Up Next Attempt}
    TimerRUP.Stop()
    If CEH::AutoList = true
        Send <mrk_rup2, i, list.Clear()>
    Else
        Send <mrk_rup, i>
    End If
    TimerRUP.Start()
```

Should be noted that when $AutoList = true$ the *RUP.GD* algorithm is sensitive to failures. Really, any failure while given process executes ring checkup leads to incomplete *LisPIds* and therefore to impossibility for subsequent graceful degradation. Therefor to prevent such incompleteness graceful degradation should be forbidden during *RUP.GD* procedure, as provided in *CEH.GD::OnOutputError* event handler (ALGORITHM 1.2).

CONCLUSION

Three working *recovery schemes* are considered within the overall project "*Class of Fault-Tolerant Distributed Algorithms for Mutual Exclusion over Elastic Logical Ring Topology*". They evolve consistently and complement each other: *Scheme 1* (Failure Recovery Without Reconfiguration), *Scheme 2* (Failure Recovery with One-way Reconfiguration) and *Scheme 3* (Failure Recovery with Two-way Reconfiguration). In this article the emphasis is on *Scheme 2*.

The consideration is made by the evolving from one scheme to another. The chain of helper distributed algorithms, as part of the modified distributed mutual exclusion algorithm, namely *Mx1ME*, is presented. With appropriate modifications of the first two of them (the *CEH* and the *RUP*) the possibility of *one-way* reconfiguration is achieved. The generalized formal specifications of those two algorithms are presented. They are consistent with the reactive character of the distributed systems and therefor consist of the handlers of events caught.

The algorithm *test bed* is implemented with the *Embarcadero C++ Builder*® development environment and its *Clang-enhanced C++* compiler. The evolution of the test bed can be traced on the author's *GitHub* repository [8].

The implementation is focused on the exploration of the algorithm as well as on the promotion of its practical usage. All working recovery schemes proposed are incorporated and could be explored separately or jointly.

Due to the features of the socket mechanism of communications and the setup configuration provided there are two possible *modes* of test bed execution: *LDM* (Local Distributed Mode) or *RDM* (Real Distributed Mode). The *LDM* is used during development, while *RDM* – after deployment. Mapping from *LDM* to *RDM* is isomorphic [7].

The discussions of the last recovery scheme (*Scheme 3*) as well as the experimental results obtained and timing analysis are subject of subsequent publications because of the space restrictions given.

REFERENCES

[1] Banerjee, S., P. Chrysanthis. (1996). A New Token Passing Distributed Mutual Exclusion Algorithm. In Proceedings of the Intl. Conf. on Distributed Computing Systems (ICDCS). Retrieved

Nov. 30, 2021, https://www.researchgate.net/publication/2666403_A_New_Token_Passing_Distributed_Mutual_Exclusion_Algorithm

[2] Christian, C., R. Guerraoui, L. Rodrigues. (2011). Introduction to Reliable and Secure Distributed Programming. 2nd Ed. Springer-Verlag Berlin Heidelberg, p. 386 pages, ISBN 978-3642152597.

[3] Coulouris, G., et al. (2011). Distributed Systems: Concepts and Design. 5th Ed. – Boston: Addison-Wesley, p. 1008.

[4] Kleinrock, L. (1985) Distributed Systems. Communications of the ACM, Vol. 28, Num. 11, pp. 1200-1213.

[5] Kshemkalyani, A.D., M. Singhal. (2008). Distributed Computing: Principles, Algorithms, and Systems. – Cambridge: Cambridge University Press, p. 736.

[6] Loukantchevsky, M. (2014). Distributed Systems: Theory and Practice. Ruse University Press, Ruse, p. 212, ISBN 978-619-7071-35-1.

[7] Loukantchevsky, M. (2020). Distributed Ring-based Mutual Exclusion with Failure Recovery. Proceedings of 21-th International Conference on Computer Systems and Technologies. ACM, New York, NY, USA, 2020, pp. 111-115, ISBN 978-1-4503-7768-3, DOI: https://doi.org/10.1145/3407982.3408014

[8] Loukantchevsky, M. (2021). XME Ring GitHub Repository. Retrieved Nov. 30, 2021, https://github.com/milphaser/XME.Ring

[9] Malkhi, D. Concurrency: the Works of Leslie Lamport. (2019). ACM, New York, NY, USA, 2020, p.366, ISBN:978-1-4503-7270-1, DOI: https://doi.org/10.1145/3335772

[10] Mohammed, A., R. Kavuri, N. Upadhyaya. (2012). Fault tolerance: case study. Proceedings of the Second International Conference on Computational Science, Engineering and Information Technology, October 2012, pp. 138–144, DOI: https://doi.org/10.1145/2393216.2393240

[11] Parihar, A.S., S.K. Chakraborty. (2021). Token-based approach in distributed mutual exclusion algorithms: a review and direction to future research. The Journal of Supercomputing, Springer Nature, May 2021, pp. 1-51, DOI: https://doi.org/10.1007/s11227-021-03802-8

[12] Ralston, A., E. Reilly, D. Hemmendinger. (2003). Encyclopedia of Computer Science. 4th Ed. - Chichester: John Wiley and Sons Ltd. , ISBN: 978-0-470-86412-8, p. 2080.

[13] Raynal, M. (1991). A simple taxonomy for distributed mutual exclusion algorithms. ACM SIGOPS Operating Systems Review, Volume 25, Issue 2, April, pp. 47-50.

[14] Saxena. P.C., J. Rai. (2003). A survey of permission-based distributed mutual exclusion algorithms. Computer Standards & Interfaces, Volume 25, Issue 2, 2003, pp. 159-181, ISSN 0920-5489, DOI: https://doi.org/10.1016/S0920-5489(02)00105-8.