Byte-Range Asynchronous Locking in Distributed Settings

Martin Quinson, Flavien Vernier

Byte-Range Asynchronous Locking in Distributed Settings

Martin Quinson
ESIAL UHP Nancy-I - LORIA
Email: martin.quinson@loria.fr

Flavien Vernier
Savoie University / LISTIC
Email: Flavien.Vernier@univ-savoie.fr

Abstract

This paper investigate a mutual exclusion algorithm on distributed systems. We introduce a new algorithm based on the Naimi-Trehel algorithm, taking advantage of the distributed approach of Naimi-Trehel while allowing to request partial locks. Such ranged locks offer a semantic close to POSIX file locking, where threads lock some parts of the shared file. We evaluate our algorithm by comparing its performance with to the original Naimi-Trehel algorithm and to a centralized mutual exclusion algorithm. The considered performance metric is the average time to obtain a lock.

1. Introduction

In multi-processes settings, shared resources must be protected against concurrent modifications to ensure data consistency. This *mutual exclusion* problem has been well studied in shared-memory environments. However, most solutions are not applicable to distributed settings since they rely on shared variables.

In distributed settings, mutual exclusion has to be provided by protocols relying on message exchanges. Several algorithms have been proposed in the literature to serialize access to a shared resource. They can be sorted in two groups: *permission-based* (e.g. Ricart-Agrawala [10]) and *token-based* (e.g. Raymond [9], Naimi-Trehel [8]). The principle of the first group of algorithms is that a node can enter the critical section (CS) only after receiving the permission of the other nodes. The drawback of this approach is the high communication overhead. In the algorithms of the second group, the right to enter the CS is granted by the possession of a token, which is unique in the system and passed over the nodes. Several algorithms of this group exhibit a $O(\log N)$ message complexity using tree-based approaches [8], [9].

We extend the Naimi-Trehel algorithm [8] in two ways. First, we assume that the protected resource is an array of elements and allows locks on subranges. This semantic is close to the POSIX file locking, where threads lock some consecutive parts of the file. This synchronization schema is useful for high performance computing [4].

The other extension we propose is inspired from [3] and consists in asynchronous locks. Applications can declare locks without immediately trying to acquire them, and continue to proceed while the locking facility acquires the needed grants from other nodes. Such asynchronous locks are not part of the POSIX standard, but they are useful in a distributed setting, where transferring the data associated to the lock over the network can be time-consuming. Asynchronous locking thus eases the overlapping of communication and computation.

To our knowledge, it is the first time that the Naimi-Trehel algorithm is extended to offer asynchronous ranged locks. [12] extends this algorithm for Read/Write-locks (provided that the duration of the critical sections is constant). [11] provides solution for fault tolerance to the Naimi-Trehel algorithm, extending the work of [7], [6]. [1] proposes an extension for hierarchical heterogeneous systems, at the price of introducing possible process starvation. [5] introduces an extension allowing different priorities for nodes. It should be possible to extend the algorithms presented here for read/write locks, fault-tolerance, heterogeneity and request priority, but this is beyond our scope.

Naimi-Trehel is based on a distributed waiting queue along which the token circulates, and on a distributed tree structure which root is the queue tail. We present an extension to this algorithm for ranged locks. The new algorithm splits the queue in partial queues when partial locks are requested. Within each queue, only the head node can be granted the lock, and the concurrency appears between the several sub-queues.

We assume that nodes handle messages in FIFO order, and that the system is failure free.

The rest of this paper is organized as follows: §2 presents the semantic of asynchronous ranged locks. §3 presents the Naimi-Trehel algorithm. Our extension is then presented in §4 and evaluated in §5.

2. Asynchronous Ranged Locking

The locking overlay requires that messages are handled at any time. Since the application may naturally

Algorithm 1: Application Interface.

Local Data:
  servant *(peer):* Locking facility process

Function `lockCreate`(*R* ∈ *resources*, *range* ∈ *P*(*r*)) ↦ *lockID*
  send (servant,"lockCreate", R, range)

Function `lockTest`(*lockID*) ↦ *bool*
  return rpc (servant, "lockTest")

Procedure `doLock`(*lockID* ↦ *data*)
  send (servant, "lockMe", lockID)
  waitMsg (expeditor=servant, type="proceed")

Procedure `unlock`(*lockID, data*)
  send (servant, "unlock", lockID, data)

block when waiting for a lock or using it, the locking facility cannot be part of the application itself and should be placed in separate thread or process. We thus separate the *application* using the locking facility from the *servant* providing this facility.

Algorithm 1 presents the servant interface. Since locks are asynchronous, their creation (with `lockCreate`) is not blocking. The request sent to the servant contains the resource's name and the subrange to lock. The application can use `lockTest` to test whether trying to acquire the lock (with `doLock`) would block. Locks can be canceled with `unlock`, even if they were not yet acquired.

The data are received along with the lock to allow their modification. Indeed, the data used by the high performance computing applications we target can be quite large, and we have to take their size into account.

This interface can trivially be used for synchronous locking: `unlock` remains unchanged while `lock` simply chains `lockCreate` and `doLock`.

The locking semantic is that locks only apply on a part of the resource designed by the range. The resource is considered as a sequential array, and it is possible to serve several locks at the same time provided that their range do not overlap. When the ranges overlap, requests are served in a FIFO mode.

3. Naimi-Trehel Algorithm

Algorithm 2 presents an asynchronous version of the base Naimi-Trehel algorithm for distributed mutual exclusion (detailed in [8]). Only regular locks are considered; range locking will be introduced latter.

In this algorithm, nodes are granted to enter the critical section if they have the token. Unsatisfied requests are stored in a distributed waiting queue, which is simply-listed: each node only knows its direct successor (*i.e.* the node to which it should pass the token afterwards). This is the next variable of each node. A node

Algorithm 2: Naimi-Trehel algorithm.

Local Data:
  owner *(peer)*
  next *(peer)*
  request *(bool)*
  token *(bool)*
  appliBlocked *(bool)*
  appli *(peer)*

Callback `lockCreateCB`(*range, mode*)
  request ← *true*
  if *(owner* ≠ *self)* **then**
    send (owner, "forwardRequest", self)
  owner ← self

Callback `forwardRequestCB`(requester*: peer*)
  if *(*token*)* **then** next ← requester
  else send (owner,"forwardRequest",requester)
  owner ← requester
  `releaseToken()`

Callback `GiveTokenCB`()
  token ← *true*
  `unblockAppli()`; `releaseToken()`

Callback `lockMeCB`()
  appliBlocked ← *true* ; `unblockAppli()`

Callback `testLockCB`()
  return token ∧ request

Callback `unlockCB`()
  request ← *false* ; `releaseToken()`

Procedure `releaseToken`()
  if *(*token ∧ ¬request ∧ next ≠ ∅*)* **then**
    send (next, "GiveToken")
    token ← *false* ; next ← ∅

Procedure `unblockAppli`()
  if *(*appliBlocked ∧ token ∧ request*)* **then**
    send (appli, "proceed")
    appliBlocked ← *false*

requesting the lock should contact the node currently at the queue's tail to enter the queue. Keeping the tail location information globally up-to-date would imply a broadcast (*i.e.* $O(n)$ messages per lock).

Instead, nodes are arranged in a distributed tree for which the root is the node placed at the tail of the queue (tree invariant). If A is the child of B in that tree, then A considers B as the queue's tail. It will send any lock requests to B. They are forwarded along the tree edges until the real tail of the queue is reached. This provides a decentralized knowledge of the tail position. The father of node A is owner from A's point of view (even if it is only the *probable* owner).

The main idea of the Naimi-Trehel algorithm is that the tree gets updated on the fly during the request propagation. When A forwards a request on B's behalf,

it means that B enters the queue. It is thus the new tail of the queue, so A sets B as its new owner.

Initially, no node requests the token, one node has the token and is considered as the owner by all nodes.

4. Split Waiting Queues (SWQ) algorithm

The SWQ algorithm can be seen as the Naimi-Trehel algorithm in which the token that represents a global resource can be split to lock the resource partially. Each sub-token represents a range of the global resource. These sub-tokens are managed like the original Naimi-Trehel token. According to the SWQ algorithm, a sub-token can be split if a sub-part of its range is requested. Respectively two continuous sub-tokens can be merged to give only one sub-token. In the following, the terms of tokens and sub-tokens are interchangeable.

We now detail our modifications to the Naimi-Trehel algorithm. The owners and nexts variables are changed from scalar values to lists. For example, nexts indicates the peers to which I should pass the lock after use. This list may contain several elements if several peers have requested for separate subranges. owned and request are changed from boolean to a list of ranges for which the condition holds. Two new variables are introduced: requestReceived lists the parts of the requested token we already received while waitingRequest lists the received requests that conflict with my own request.

The messages exchanged between servant to request the token contain the requester id, the requested range (noted request) and the range already found.

As in the original algorithm, if a node does not know the owner of tokens, it is itself the owner.

4.1. The SWQ Algorithm details

To keep this presentation simple, we assume that a given node only requests one continuous range at a given time, and release its token before the next request.

In the SWQ Algorithm, a request is proceeded sequentially - the first part of the request is looked for, then the subsequent part is looked for, and so on - such as not to generate a dead lock. Oppositely to the classical Naimi-Trehel algorithm, if a node requests a range, it doest not immediately become the owner of this range. It only does so when its request message running the network finds a part of the request. This mechanism avoids some dead-lock or loop configurations.

The SWQ algorithm can be decomposed into three steps: the initial request, the reception and forwarding requests, and the exchange of tokens. These steps are achieved by the functions given in Algorithm 3.

The algorithm starts when a node asks a part of the token. It checks if is the owner of the first part

Algorithm 3: Split Waiting Queue.

```
Local Data:
    nexts (tokensList)
    owners (tokensList)
    owned (tokenList)
    request (token_t)
    requestReceived (tokensList)
    waitingRequest (requestsList)
Procedure requestToken()
    found ← Search First Part of request
    if (found != null ∧ found ∈ owned) then
        add found into requestReceived
    if (found != request) then
        nextOwner ← Search Next Owner of found
        requestMsg ← new message (request,
        found, mySelf)
        send (nextOwner, "requestToken",
        requestMsg )
Callback requestTokenCB(requestMsg: request_t)
    found ← Search Next Part(requestMsg)
    if !conflict() ∨ requestIsSatified() then
        if (found != NULL) then
            update(requestMsg, found)
            if (mySelf != requestMsg.requester) then
                add found into nexts
                send (requestMsg.requester, "foundPart",
                found )
            else if (found ∈ owned) then
                add found into requestReceived
        if (requestMsg.found != requestMsg.request)
        then
            nextOwner ← Search Next Owner of
            requestMsg.found
            send (nextOwner, "requestToken",
            requestMsg )
        if (mySelf != requestMsg.requester) then
            update owners with found
    else
        add requestMsg into waitingRequest
Callback foundPartCB(token: token_t)
    remove token from owners
Procedure sendTokens()
    foreach element in nexts do
        if element.token ∈ owned ∧ do not use
        element.token then
            send (element.next, "exchangeToken",
            element.token)
            remove element.token from nexts
            remove element.token from owned
Callback exchangeTokenCB(token: token_t)
    add token into owned
    add token into requestReceived
```

of its request (see requestToken in Algorithm 3). If it finds this part, it updates the already found range of the request, and if it owns this token, it updates its requestReceived list. Let us recall that a node can be the owner of a token that it has not received yet. If it is not the owner of all its requested range it sends a request message to the owner of the part that follows the already found part.

When a node receives a request (see requestTokenCB), it checks if it is the owner of the searched part of this request. If it finds a part and this part is not in conflict with its own request, it updates the already found range of the request message and it sends the information about the found part to the original requester. If it is not the original requester, it updates its nexts and owners lists. In all cases, if the request is not satisfied, it forwards this request to the supposed owner of the searched part of the request.
A particular case can appear if this node is the original requester. In this case the node does not send the information about the found part, it directly updates its requestReceived list if it has the token and does not update its owners list.
If the found part is in conflict with its own request, the request message is stored into the waitingRequest list and will be proceeded when the local node is the owner of its own request.

The other actions performed by the algorithm - the management of information messages and tokens exchange - are less complex. When a node receives the information about a found part of its request, it updates its owners list. It becomes the new owner of this part. If the nexts list of a node is not empty, it sends each token of this list to the requester if it has the token and does not use it, and it updates its owned and nexts lists if necessary. When a node receives a token, it updates its owned list and its requestReceived list.

α	range	algo	# of nodes 2	12	22	42	82	162
0.25	$\frac{1}{16}$	NT	0.8	13.1	23.4	44.7	82.6	165
		central	0.2	1.6	3.6	7.6	16.3	33.5
		SWQ	0.3	3	5	11	32.6	117
	full	NT	0.8	13.1	23.4	44.7	82.6	165
		central	1	19.1	37.8	70.2	138	277
		SWQ	0.9	26.6	72.1	228	745	2807
1	$\frac{1}{16}$	NT	1.4	21.2	39	75.1	143	284
		central	0.2	1.5	3.3	7.2	15.7	33.3
		SWQ	0.3	3.1	5.1	12.3	30.3	116
	full	NT	1.4	21.2	39	75.1	143	284
		central	1.6	27.6	53.8	101	199	396
		SWQ	1.4	26.9	72.1	228	747	2810
5	$\frac{1}{16}$	NT	4.5	64.8	122	237	463	922
		central	0.6	5.1	9.7	18.9	37.2	73.1
		SWQ	0.6	6.2	13.1	31.3	60.9	153
	full	NT	4.5	64.8	122	237	463	922
		central	5.7	71.7	137	264	520	1035
		SWQ	4.5	64.8	122	268	787	2860

Table 1. Average timings in seconds (size=64kb).

5. Experimentations

The experiments were run on the SimGrid simulator [2]. We compare the Naimi-Trehel algorithm, Centralized range token management and the SWQ algorithm. In each experiment, every node requests 25 locks.

We study the impact of the following parameters: the time each lock is kept by nodes (denoted α), the fraction of the resource locked (denoted range), the number of nodes and the size of the resource.

Table 1 presents the timing of the studied algorithms for representative values α and range when the resource size is fixed to 64kb (8192 double values). Since the Naimi-Trehel algorithm does not consider ranges, its timing are the same for the different values of range.

The first result is that the Naimi-Trehel algorithm leads to smaller waiting times than the centralized implementation when the full range is requested. For example, when $\alpha = 1$ for 162 nodes, locks are obtained in 284s on average with NT *vs.* 396s with centralized. This result is not new, but is clearly highlighted here.

Conversely, when only parts of the resource are locked, NT leads to bigger waiting time than centralized. For example, when $\alpha = 1$ for 162 nodes, locks are still obtained in 284s on average with NT while centralized now achieves 33.3s waiting times. This clearly demonstrates the potential benefit of partial locking, since it shows that the whole benefit of distributed locking from NT is nullified by the optimization made possible by partial locks.

When comparing SWQ to NT, one can remark that it is always slower than the classical Naimi-Trehel algorithm on locks requesting the full resource. This is because of the guards we added to manage the split tokens. Thus, although the global behavior of these two algorithms is the same with global requests, the SWQ algorithm is slowed down by its split token management ability. The benefit of SWQ becomes clear when considering partial locks. For example when $\alpha = 1$ for 162 nodes, NT achieves 284s waiting times for partial

size	α	algo	# of nodes 2	12	22	42	82	162
640kb	0.25	central	0.4	5.3	11.5	21.8	43.9	86.4
		SWQ	0.5	4.3	6.4	14.2	36	124
	1	central	0.5	5	11	21.2	42.9	87.1
		SWQ	0.5	4.3	6.7	15	36.6	121
	5	central	0.7	7	13.4	24.4	47.7	92.9
		SWQ	0.7	7.3	14	32.3	66.4	151
6.25Mb	0.25	central	4.9	48.2	92.8	165	318	618
		SWQ	2.6	21	27.1	50.8	91.8	199
	1	central	4.9	47.4	91.9	163	315	619
		SWQ	2.8	20.2	27.8	49.8	88	201
	5	central	4.9	46	88.7	159	312	618
		SWQ	2.6	19.9	30.3	66	113	240

Table 2. Impact of size on timings $\left(\text{range} = \frac{1}{16}\right)$.

locks while SWQ achieves 116s waiting times in the same settings.

The comparison of SWQ to centralized is not very flattering for our algorithm: in the presented experiments, centralized is always faster than SWQ. Table 2 compares the timings of centralized and SWQ when increasing the resource size. Former experiments were done with a resource of 64kb, we now use 640kb and 6.25Mb resources (respectively 81,960 and 819,600 double values). The range is kept to $\frac{1}{16}$ here.

These results show the huge impact that the resource size have on the waiting times. When size=640kb, centralized still outperform SWQ, but not as much as before: For $\alpha = 1$ for 162 nodes, centralized is more than three time faster than SWQ for 64kb, and "only" 30% faster for 640kb. When size=6.25Mb, SWQ is three time faster than centralized.

This can be explained by the fact that the data is transfered to the requesting applications along with the lock. Centralized thus struggles with the amount of data exchanged by its central point. So, for larger resources, the cost of our distributed design is smaller than the gain of its lack of central bottleneck.

These results highlight the benefits of SWQ for partial locking of large resources.

6. Conclusion

This paper introduces a new distributed mutual exclusion algorithm presenting two main specificities: it allows to partially lock the resource and it manages the resource and the locks without any central point. The proposed algorithm should reveal useful in context of grid computing, where distributed processors handle large data sets, making centralized management of the mutual exclusion penalizing or even impossible. This is the case for systems with geographically distributed clusters. Experimental results show that in such contexts, our decentralized algorithm reduces the latency to acquire a critical section and thus improves the overall performance of parallel iterative algorithms.

Moreover the proposed algorithm allows asynchronous locks. Applications can declare locks without immediately trying to acquire them, and continue to proceed while the locking facility acquires the needed grants from other nodes. Such asynchronous locks are not directly part of the POSIX standard, but are useful in a distributed setting, where transferring the data associated to the lock over the network can be time-consuming. This semantic extension allows the data to be sent in advance to the requester.

We foresee two directions of future work: First, we plan to compare the presented simulation results to live deployments of our program. Then, we would like to relax some constraints that we introduced for the management of conflicts between split queues so that our performance becomes closer to the Naimi-Trehel ones for full resource locking.

References

[1] L. Arantes, M. Bertier, and P. Sens. Distributed mutual exclusion algorithms for grid applications: A hierarchical approach. *J. of Parallel and Distributed Computing*, 66:128–144, 2006.

[2] H. Casanova, A. Legrand, and M. Quinson. SimGrid: a Generic Framework for Large-Scale Distributed Experiments. In *10th IEEE Intl Conference on Computer Modeling and Simulation*, 2008.

[3] J. Gustedt. Data HandOver: Reconciling message passing and shared memory. In *Foundations of Global Computing*, 2005.

[4] J. Gustedt, S. Vialle, and A. De Vivo. parXXL: A Fine Grained Development Environment on Coarse Grained Architectures. In *Workshop on State-of-the-Art in Scientific and Parallel Computing - PARA'06*, Sweden, 2006.

[5] F. Mueller. Prioritized token-based mutual exclusion for distributed systems. In *Intl Parallel Processing Symposium*, pp 791–795, 1998.

[6] F. Mueller. Fault tolerance for token-based synchronization protocols. *IEEE Workshop on Fault-Tolerant Parallel and Distributed Systems*, 2001.

[7] M. Naimi and M. Trehel. How to detect a failure and regenerate the token in the log(n) ditributed algorithm for mutual exclusion. In *LNCS*, 312:155–166, 1987.

[8] M. Naimi, M. Trehel, and A. Arnold. A log(n) ditributed mutual exclusion algorithm based on path reversal. *J. of Parallel and Distributed Computing*, 34(1):1–13, 1996.

[9] K. Raymond. A tree-based algorithm for distributed mutual exclusion. *ACM Transaction on Computer Systems*, 7(1):61–77, 1989.

[10] G. Ricart and A. Agrawala. An optimal algorithm for mutual exclusion in computer networks. *Communication of the ACM*, 24, 1981.

[11] J. Sopena, L. Arantes, M. Bertier, and P. Sens. A fault-tolerant token-based mutual exclusion algorithm using a dynamic tree. In *Euro-Par'05 - Parallel Processing*, LNCS, 2005. Springer-Verlag.

[12] C. Wagner and F. Mueller. Token-based read/write-locks for distributed mutual exclusion. In *European Conference on Parallel Processing*, pp 1185–1195, 2000.