Distributed Hierarchical Locking with Real-Time Priorities *

Nirmit Desai and Frank Mueller
Department of Computer Science
North Carolina State University, Raleigh, NC 27695-7534
mueller@cs.ncsu.edu, phone: +1.919.515.7889, fax: +1.919.515.7925

Abstract

On large distributed systems, computational resources are shared by autonomous and independent applications. Scalability for synchronization in such environments can be achieved by supporting hierarchical locking as we have described in [3]. However, the applications accessing the shared resources may have stringent response time requirements to function correctly. Previous approaches for scalable concurrency services do not address this problem in the realm of real-time applications.

This work contributes design and implementation details of a novel peer-to-peer protocol for distributed concurrency services with support for real-time systems. Our protocol realizes resource arbitration through hierarchical locking, it is highly scalable for large distributed systems and it bounds priority inversion. We address the problem of priority inversion by supporting two of the most prominent inheritance protocols: PCEP and PIP. Experiments confirm the scalability of our protocol in conjunction with its support for priorities. Prioritized requests are shown to result in predictable and bounded response times. Support of PCEP bounds priority inversion but may incur false blocking. While the PIP solution does not suffer from false blocking, its bounds on priority inversion include additional message overhead. Our work impacts real-time applications sharing resources across large distributed environments ranging from hierarchical locking in real-time databases and database transactions to distributed object environments in large-scale embedded systems.

1. Scalable Hierarchical Locking

We first introduce our base protocol to provide hierarchical locking in a distributed environment [3] and then extend it by support for priority queuing. Systems generally require the distinction between lock modes to increase the potential for concurrency. Most systems distinguish read (R) locks and write (W) locks with parallel and exclusive access, respectively. An upgrade (U) lock represents an exclusive read lock that is followed by a request for promotion to a write lock. Upgrade locks avoid conflicts between threads that intend to modify (write) objects after having held a read lock. Hierarchical locking schemes enhance parallelism and reduce locking overhead by distinguishing lock modes based on the structure of data representation. Entities at the higher level of granularity are associated with intention versions (IR and IW) of the aforementioned lock modes (R and W).

In our protocol, nodes form a logical tree structure by maintaining their local parent pointers. The root node has the token and is referred to as the token node, all other nodes are referred to as non-token nodes. We briefly describe the Rules below, governing the workings of the protocol:

*This work was supported in part by NSF grant CCR-0237570.

Rule 1: Modes M_1 and M_2 are said to be *compatible* with each other if and only if they are not in conflict according to compatibility matrix of Table 1(a) and [4].

Definition 1: Lock A is said to be *stronger* than lock B if the former constrains the degree of concurrency over the latter. In other words, A is compatible with fewer other modes than B is. The order of lock strengths is defined by the following inequations:

$$\Phi < IR < R < U = IW < W \quad (1)$$

Definition 2: Node A is said to *hold* the lock L_R in mode M_H if A is inside a critical section protected by the lock, *i.e.*, after A has acquired the lock and before A releases it.

Definition 3: Node A is said to *own* the lock L_R in mode M_O if M_O is the strongest mode being held by any node in the tree rooted in node A.

Rule 2: A node sends a request for the lock in mode M_R to its parent if and only if the node owns the lock in mode M_O where $M_O < M_R$ (and M_O may be Φ), or M_O and M_R are incompatible. Otherwise, the local copyset is updated and the critical section is entered without messages.

Rule 3:

1. A non-token node owning L_R in mode M_O can grant a request for L_R in mode M_R if M_O and M_R are compatible and $M_O \geq M_R$.
2. The token node owning L_R in mode M_O can grant a request for L_R in mode M_R if M_O and M_R are compatible.

The operational protocol specification further requires:

in case 1: the requester becomes a child of the node;
in case 2: if modes are compatible and if $M_O < M_R$, the token is transferred to the requester. The requester becomes the new token node and parent of the original token node. If $M_O \geq M_R$, the requester receives a granted copy from the token node and becomes a child of the token node. (See Rule 4 for the case when M_O and M_R are incompatible.)

Tab. 1(b) depicts legal owned modes M_1 of non-token nodes for granting a mode M_2 according to Rule 3, indicated by the absence of an X.

Rule 4:

1. If a non-token node with pending requests for mode M_1 cannot grant a new request for M_2, it either forwards (F) the request to its parent or locally queues (Q) according to Table 1(c).
2. If the token node cannot grant a request, it will queue the request locally regardless of its pending requests.

Rule 4 is supplemented by the following operational specification: Locally queued requests are considered for granting when the pending request comes through or a release message is received.

Rule 5:

1. When the token node releases a lock or receives release from one of its children, it considers the locally queued requests for granting under constraints of Rule 3.

Owned mode M_O:	(a) Incompatible					(b) No Child Grant					(c) Queue/Forward					(d) Freezing Modes at Token				
Request mode M_R	IR	R	U	I	W	IR	R	U	I	W	IR	R	U	IW	W	IR	R	U	I	W
No lock – Φ						X	X	X	X	X	F	F	F	F	F					
Intent Read – IR					X		X	X	X	X	Q	F	F	F	F					IR,R,U,IW
Read – R				X	X			X	X	X	F	Q	F	F	F				R,U	IR,R,U
Upgrade – U			X	X	X			X	X	X	F	F	Q	Q	Q				R	IR, R
Intent Write – IW		X	X		X		X	X		X	F	F	F	Q	F		IW	IW		IR, IW
Write – W	X	X	X	X	X	X	X	X	X	X	Q	Q	Q	Q	Q					

Table 1: Rules of Hierarchical Locking for Mode M_1 relative to Mode M_2

2. When a non-token node A releases a lock or receives a release in some mode M_R, it will send a release message to its parent only if the owned mode of A is changed (weakened) due to this release.

Rule 6: A node may only grant a request if the requested mode is not frozen.

This rule supplements Rules 2 and 3. Tab. 1(d) depicts the frozen modes for all combinations.

Rule 7: Upon an attempt to upgrade to W, the token owner atomically changes its mode from U to W (without releasing the lock in U).

2. Prioritized Scalable Hierarchical Locking

We next describe the prioritized version of our protocol. Let $\beta(N)$ and $\xi(N)$ be the base and effective priorities of node N, respectively. Each request made by node N has priority $\xi(N)$, the effective priority of N at the time of request. Without the support for a priority inheritance scheme, $\xi(N)$ and $\beta(N)$ would not be distinguished.

The basic approach of supporting prioritized requests is to redesign the

1. **Freezing Mechanism:** Overrule the frozen modes if the received request has higher priority than the request waiting in queue.
2. **Queue Reordering:** Order the local queue of requests first according to their priority $\xi(N)$ and then according to FIFO.
3. **Queue/Forward Policy:** Change the local queuing policy to take into account the priority of received request vs. the priority of pending self-requests in case of Rule 4.1.

We describe each of these improvements and their motivations in the following subsections.

2.1 Freezing Mechanism

The current implementation of the freezing mechanism potentially results in potentially unbounded priority inversion. The rationale behind the construction of the Table 1(d) can be formalized by the following invariants:

INVARIANT 1. *Let M_{FT} be the set of modes to be frozen at token node when the token node owning lock L_i in mode M_{OT} receives a request for L_i in mode M_R. Then,* $\forall M_i \epsilon M_{FT} : [grantable(M_{OT}, M_i) \vee tokenable(M_{OT}, M_i)] \wedge conflicts(M_i, M_R)$.

INVARIANT 2. *Let M_{FC} be the set of modes to be sent with the freeze message to a child owning L_i in M_{OC}. Then,* $\forall M_i \epsilon M_{FC} : M_i \epsilon M_{FT} \wedge grantable(M_{OC}, M_i)$.

In the basic protocol, the only values each of the elements in $Frozen\_Modes$ can take are $FROZEN$ and NOT_FROZEN. Rule 6 uses this information and approves a grant/token for requested mode M_R if the set $Frozen\_Modes$ has NOT_FROZEN as the state value of the element corresponding to M_R. This means that any request, regardless of its priority, can cause a set of elements to assume $FROZEN$ state if it is queued at the token node. Any request for any of those modes in the future, regardless of their priority, is disapproved by Rule 6. The solution to the problem is quite intuitive and elegant. Let *threshold* be the priority of the highest priority request causing a freeze. The elements of the $Frozen\_Modes$ set record the *threshold* of the requests causing the freeze. Tab. 2 is the prioritized version of Tab. 1(d) for the basic protocol.

Token Node owning Mode M_{OT}	Requested Mode M_R				
	IR	R	U	IW	W
No lock – Φ					
Intention Read – IR					IR, R, U, IW
Read – R				R, U	IR, R, U
Upgrade – U				R	IR, R
Intention Write – IW		IW	IW		IR, IW
Write – W					
$\forall M_i \epsilon Frozen\_Modes : Frozen\_Modes(M_i).threshold = max\,[Frozen\_Modes(M_i).threshold,\ \xi(M_R)]$					

Table 2: Freezing Lock Modes at the Token Node

Rule 6 can be replaced by its prioritized version as:

Rule 6_{PRIO}: A request for mode M_R can only be granted if $\xi(M_R) > Frozen\_Modes(M_R).threshold$.

2.2 Reordering Queue

The following rule gives preference to higher priority requests when requests are queued locally.

Rule 8:

1. While queuing a request with priority $\xi(N)$ locally, it is queued first according to the non-increasing order of priority and then according to non-increasing order of *usage times* and finally increasing order of their *receipt times* (described in [5, 6]).
2. While receiving the token and merging the local queue with the queue received from the original token node, the queue is sorted according to the order described in Rule 8.1.

2.3 Queue/Forward Policy

The essence of the policy to queue requests locally is to enable path compression for requests, thereby reducing the message overhead without violating FIFO.

Consider real-time applications with $\xi(D) > \xi(B)$. According to Rule 8.1, $\{D, R\}$ will precede $\{B, R\}$ in the queue at A. This means that queuing $\{D, R\}$ locally at B results in (potentially unbounded) priority inversion. This leads us to extend the Table 1(c) as shown in Table 3, bold faces indicating extensions. Essentially, to queue the request locally, $\xi(pending\_request) \geq \xi(received\_request)$ should be met in conjunction to the conditional of Table 1(c).

Non-token Pending Mode M_P	Requested Mode M_R				
	IR	R	U	IW	W
No pending – Φ	F	F	F	F	F
Intention Read – IR	**CQ**	F	F	F	F
Read – R	F	**CQ**	F	F	F
Upgrade – U	F	F	**CQ**	**CQ**	**CQ**
Intention Write – IW	F	F	F	**CQ**	F
Write – W	**CQ**	**CQ**	**CQ**	**CQ**	**CQ**
Conditional Queuing CQ = IF($\xi(M_P) \geq \xi(M_R)$) THEN 'Q' ELSE 'F'					

Table 3: Queue / Forward Decision

3. Priority Inheritance

Two basic methods for bounding priority inversion can be incorporated into the prioritized protocol. First, early priority boosting can be discussed in the context of the priority ceiling emulation protocol (PCEP). Second, the protocol can be enhanced for dynamic priority changes due to resource contention to support the priority inheritance protocol (PIP). For both cases, changes to the prioritized protocol are identified.

Priority Ceiling Emulation Protocol: Priority ceiling protocols in general require that a priority values, the *ceiling* C_i, be associated with a resource R_i and the corresponding lock L_i. C_i is defined as the priority of the highest priority node contending for L_i. The priority ceiling emulation protocol (PCEP) combines the general idea of the original PCP [7] with early priority boosting to the ceiling level immediately upon granting resource requests as proposed in the stack resource protocol [1]. It also requires that the ceilings be computed statically (as for PCP). When granting a resource R_i, PCEP boosts the priority of a requester node N to $max(\xi(N),\ C_i)$. When releasing a resource R_i by node N, which still holds a subset L_h of L resources, PCEP lowers the effective priority of N to $max(\beta(N),\ max(C_h))$. The protocol avoids chained blocking, priority changes while holding a resource and may reduce the number of context switches for a task on node N. On the other hand, it may experience *false blocking* in situations where the previous protocols would not have blocked. The most important advantage of PCEP is its simplicity in terms of implementation. As a result, PCEP has been adopted by industry standards (*e.g.*, POSIX and Ada95).

PCEP can be used with minimal changes to the prioritized protocol presented so far. It can be extended as (bold face extensions in Figure 1):

1. Boost the requester's priority upon acquiring the resource in *ReceiveToken* or *ReceiveGrant* as described before.
2. Lower its priority upon releasing the resource in *RequestUnlock* as described before.

These modifications do not require any additional message overhead. The resulting protocol provides the most efficient solution to bounding priority inversion in the protocols for real-time distributed concurrency control to our knowledge.

Priority Inheritance Protocol (PIP): The PIP protocol requires that the priority of a node holding a resource be boosted to some value upon detection of a contention for this resource [7]. The protocol also requires transitive boosting when node A requests resource R_i held by node B and node B has issued a request for resource R_j held by node C. This situation is commonly known as chained blocking. In this case, C will inherit A's priority (if $\xi(A) > \xi(C)$) since node

ReceiveToken()
...
ξ(**self**) $\leftarrow$ **max**(ξ(**self**), $\mathbf{C_{lock}}$)
... //All the other operations
ReceiveGrant()
...
ξ(**self**) $\leftarrow$ **max**(ξ(**self**), $\mathbf{C_{lock}}$)
... //All the other operations
RequestUnlock()
...
ξ(**self**) $\leftarrow$ **max**(β(**self**), $\mathbf{C_{held}}$)
... //All the other operations

Figure 1: Extensions for Ceiling Emulation

C causes node A to be blocked indirectly through node B. Transitive priority changes may cause a chain of priority boosts along the contention dependencies of resources.

Also, considering the global context of locks L, a node N may have its effective priority boosted due to a contention for multiple locks in L that N is holding. Hence, when N releases one of its locks, its effective priority should be the highest of the effective priorities due to all the locks it is still holding. To calculate this new effective priority, we keep track of the highest boosted priority for each of the locks and call it the effective priority of a lock $\xi(lock)$. We describe the extensions as follows (see bold-face extensions in Figure 2):

HandleRequest: If the request for M_R is being queued at the token node (a contention is detected) then

1. Send the freeze messages with frozen thresholds to its children. (This notifies the children holding conflicting modes for this lock for boosting their priority as stated in Lemma 1.)
2. If the request conflicts with token node's (self) held mode, boost self's effective priority to $\xi(self) = max(\xi(self), \xi(M_R))$ and boost the effective priority of this lock to $\xi(lock) = max(\xi(lock), \xi(M_R))$. If self's effective priority is changed, re-issue all the pending requests for all the locks $L_i \epsilon L$. (This boosts self's effective priority in case of a conflict.)

HandleFreeze: If a child receives a freeze message from its parent and if the child is *holding* the lock in any mode, a conflict between a child's held mode and a recently queued requested mode at the token node is detected. The child boosts its effective priority to $\xi(self) = max(\xi(self), max(Frozen\_Modes.threshold))$. If self's effective priority is changed, the child reissues all the pending requests for all the locks $L_i \epsilon L$. The child also boosts the effective priority for this lock to $\xi(lock) = max(\xi(lock), max(Frozen\_Modes.threshold))$.

RequestUnlock: When a lock is released, self's effective priority is lowered to $\xi(self) = max(\beta(self), max(\xi(\forall L_i \epsilon L :$ IF(Held(L_i)) THEN $\xi(L_i)$ ELSE NIL))).

Once the lock is released, the effective priority for this lock is lowered to the lowest priority NIL, *i.e.*, $\xi(lock) = NIL$.

HandleReissuedRequst: If a reissued request is received that is already present in our local queue, the queued request's effective priority is updated to the reissued request's effective priority. This may require reordering of the queue according to Rule 8.1. Also, if we are not the token node and the reissued request has higher priority than our pending request, we can no longer queue the request locally (Table 3). Hence, the request is dequeued and forwarded to the parent. Also, updating the priority of the queued

HandleRequest($\mathbf{M_R}$)
IF Self $\neq$ Token_node THEN
IF grantable(M_O, M_R) $\wedge$
Rule $\mathbf{6_{PRIO}}$ THEN [Rule 3.1]
Children $\leftarrow$ Children + Requester
Copyset $\leftarrow$ Copyset + M_R
Send grant to Requester
ELIF queuable(M_P, M_R) THEN [Rule 4.1,Tab 3]
Queue $\leftarrow$ Queue + M_R
ELSE [Rule 4.1]
Send Request to Parent
ELSE
IF tokenable(M_O, M_R) THEN [Rule 3.2]
IF Requester $\in$ Children THEN
Children $\leftarrow$ Children - Requester
Parent $\leftarrow$ Requester
Send Token to Requester
ELIF grantable(M_O, M_R) THEN [Rule 3.2]
Children $\leftarrow$ Children + Requester
Copyset $\leftarrow$ Copyset + M_R
Send Grant to Requester
ELSE [Rule 4.2]
Queue $\leftarrow$ Queue + M_R
IF conflicts($\mathbf{M_H}$, $\mathbf{M_R}$) THEN
ξ(self) $\leftarrow$ max(ξ(self), ξ($\mathbf{M_R}$))
ξ(lock) $\leftarrow$ max(ξ(lock), ξ($\mathbf{M_R}$))
IF changed(ξ(self)) THEN
re $-$ issue all the pending requests
for all the $\mathbf{L_i} \epsilon$ L
Update Frozen_Modes [**Inv 2**]
Send Freeze to Children [**Inv 3**]
HandleFreeze(Frozen_Modes)
IF $\mathbf{M_H} \neq; \phi$ THEN
ξ(self) $\leftarrow$ max(ξ(self), max(Frozen_thresholds))
IF changed(ξ(self)) THEN
re $-$ issue all the pending requests
for all the $\mathbf{L_i} \epsilon$ L
ξ(lock) $\leftarrow$ max(ξ(lock), max(Frozen_thresholds))
Send Freeze to Children [**Inv 3**]
RequestUnlock()
Copyset $\leftarrow$ Copyset - M_H
$M_H \leftarrow \phi$
IF Changed Mode of Self THEN [Rule 5]
Send Release to Parent
Check_request_on_queue [Rule 5]
ξ(self) $\leftarrow$ max(β(self), max($\forall \mathbf{L_i} \epsilon$ L : $\xi(\mathbf{L_i})$))
ξ(lock) $\leftarrow$ NIL
HandleReissuedRequest()
IF req ϵ Queue THEN
ξ(req) $\leftarrow$ ξ(new_req)
Rearrange the queue [**Rule 8.1**]
IF Self $\neq$ Token_node THEN
IF $\xi(\mathbf{M_P}) < \xi$(new_req) THEN
Dequeue and forward the request [**Tab 3**]
Update Frozen_Modes [**Table 2**]
Send freeze to children [**Inv 3**]

Figure 2: Extensions for Inheritance

request may cause the frozen thresholds to be boosted and sent to the children according to Table 2 and Invariant 2.

Notice that while sending freeze messages to children with necessary thresholds (as given by Table 2 and Invariants 1 and 2), we assume that whenever there is a potential conflict between the request being queued and the held mode of the children, a freeze message will always be sent to those children. Lemma 1 explicitly states this requirement:

LEMMA 1. *If a child owns lock L_i in mode M_{OC} and a request for mode M_R is received by the token node owning L_i in M_{OT} such that the modes are in conflict, then a freeze message will be sent to the child.*

Proof: conflicts(M_{OC}, M_R) $\Rightarrow$ conflicts(M_{OT}, M_R) [as $M_{OC} < M_{OT}$]. Let M_{FT} be a set of all modes to be frozen at the token node according to Invariant 1,

$$\forall M_i \epsilon M_{FT} : conflicts(M_i, M_R) \wedge (grantable(M_{OT}, M_i) \vee tokenable(M_{OT}, M_i)) \quad (2)$$

Let M_{FC} be a set of all modes that will be sent to the child according to Invariant 2,

$$\forall M_j \epsilon M_{FC} : grantable(M_{OC}, M_j) \wedge conflicts(M_j, M_R) \wedge (grantable(M_{OT}, M_j) \vee tokenable(M_{OT}, M_j)) \quad (3)$$

If the M_{FC} calculated in constraint 3 is non-empty, the token node will send a freeze to the child.

By substituting $M_j = M_{OC}$ in constraint 3, the constraint is satisfied as $\forall\ M_{OC} : grantable(M_{OC}, M_{OC})$ [U and W mode owners cannot be children], conflicts(M_{CO}, M_R) is true and grantable(M_{OT}, M_{OC}) is always true for a child of a token node.

$\Rightarrow M_{FC}$ is non-empty.

$\Rightarrow$ a freeze message will be sent to the child.

4. Conclusion

We have designed and implemented a novel approach for real-time distributed concurrency services. Our protocol supports hierarchical locking and it is highly scalable for large distributed systems. We also address the problem of priority inversion by supporting two of the most prominent inheritance protocols: PCEP and PIP. Experiments, omitted here due to space constraints, indicate that adding support for priorities does not affect the scalability of our protocol (see [2]). It also shows that prioritized requests have predictable and bounded response times, particularly for PIP. Support of PCEP bounds priority inversion but introduces the problem of false blocking. While the PIP solution does not suffer from false blocking, it gives results in bounds on priority inversion duration due to additional message overhead. The results of our work impact real-time applications sharing resources across large distributed environments ranging from hierarchical locking in real-time databases and database transactions to distributed object environments in large-scale embedded systems.

5. REFERENCES

[1] T.P. Baker. Stack-based scheduling of realtime processes. *Real-Time Systems*, 3(1):67–99, March 1991.

[2] N. Desai. Scalable distributed concurrency protocol with priority support. Master's thesis, North Carolina State University, June 2003.

[3] N. Desai and F. Mueller. Scalable distributed conucrrency services for hierarchical locking. In *International Conference on Distributed Computing Systems*, page (accepted), May 2003.

[4] Object Management Group. Concurrency service specification. http://www.omg.org/tech-nology/documents/formal/con-currency_service.htm, April 2000.

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

[6] F. Mueller. Priority inheritance and ceilings for distributed mutual exclusion. In *IEEE Real-Time Systems Symposium*, pages 340–349, December 1999.

[7] Lui Sha, Ragunathan Rajkumar, and John P. Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Trans. Computers*, 39(9):1175–1185, September 1990.