Prioritized Scalable Distributed Concurrency Services

Group members
Nirmit Desai: nvdesai@cs.ncsu.edu

Project web
http://www4.ncsu.edu/~nvdesai/middlepro.html

Problem description

The project had multiple objectives:

1. Extend the protocol for scalable distributed concurrency services to support priority levels of requests.
2. Experiment with the various priority levels to study the response time behavior for different levels of priority.
3. Find a mechanism/policy to bound the response time of requests based on their priorities. Experiment with keeping the concurrency level constant.
4. Make progress on the TAO front.

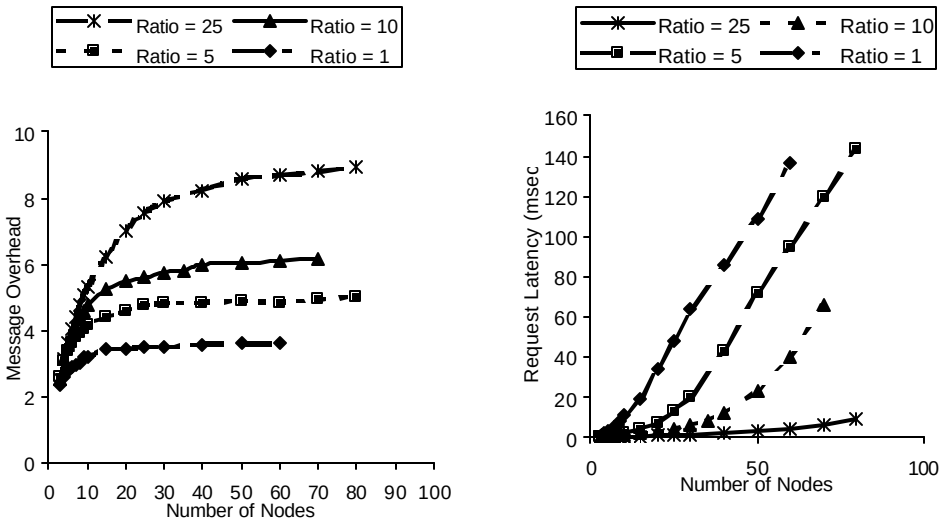
Current status of the abovementioned objectives

1. The implementation with priorities was already done previously. The description of the implementation is included at the end of the document.

2. Some experiments with the prioritized implementation are already done. They demonstrate a clear separation of response time between different kinds of requests, but they are still not bounded.

3. Bounding the response time means having logarithmic behavior for the response time as well as preserving the logarithmic behavior of the message overhead.

   Though the message overhead behavior can be used to study the behavior of the protocol, it may not represent the request latency accurately. This is due to the fact that the request latency time has two components: The network delay experienced by each of the message sent and the queuing delay due to the request being locally queued at other nodes. While the former can be accurately estimated by the message overhead, the later does not show up as part of the message overhead. This means that response time behavior can be identical in ideal case or worse than message overhead behavior. This is evident from the results shown below for a representative experimental environment. The Message overhead is logarithmic and bounded but the response time is linearly (sometimes superlinearly) increasing with the number of nodes in the system.



In this experiment, as the number of nodes in the system increases, average number of requests active at any time also increases. This causes more conflicts between incompatible request types thereby increasing the queuing delay. However, the tree height increases logarithmically and so does propagation path of requests thereby making the message overhead logarithmic. So, with the increase in number of nodes, the queuing delay increases which is added to the logarithmically increasing message overhead. The final product is the linearly increasing response time.

This also means that the increase in the queuing delay should be super-linear. The following argument may explain this:

Consider a time point during the run of the protocol. The probability of a request mode m at such a point of time is n*p(m), where n is the number of nodes in the system, and p(m) is the probability of an m request given by the randomized stream of each node. This is due to the fact that request types are randomized, and each node has an independent randomized stream. So, the probability of each of the modes of the requests increases linearly with the number of nodes in the system.

Let f(n) denote the function giving the number of nodes in the system. Clearly, f(n)=n.

Let c(n) denote the number of conflicts present at any point of time.

By the compatibility matrix and the discussion above,

C(n) =
f(n).p(W).[f(n).p(IR) + f(n).p(R) + f(n).p(U) + f(n).p(IW) + f(n).p(UW) + f(n).p(W)] +
f(n).p(UW).[f(n).p(IR) + f(n).p(R) + f(n).p(U) + f(n).p(IW) + f(n).p(UW) + f(n).p(W)] +
f(n).p(IW).[f(n).p(R) + f(n).p(U) + f(n).p(UW) + f(n).p(W)] +
f(n).p(U).[f(n).p(U) + f(n).p(IW) + f(n).p(UW) + f(n).p(W)] +
f(n).p(R).[f(n).p(IW) + f(n).p(UW) + f(n).p(W)] +
f(n).p(IR).[f(n).p(UW) + f(n).p(W)]

Where, the p(IR)+p(R)+p(U)+p(IW)+p(UW)+p(W)=1 and f(n)=n.
So,

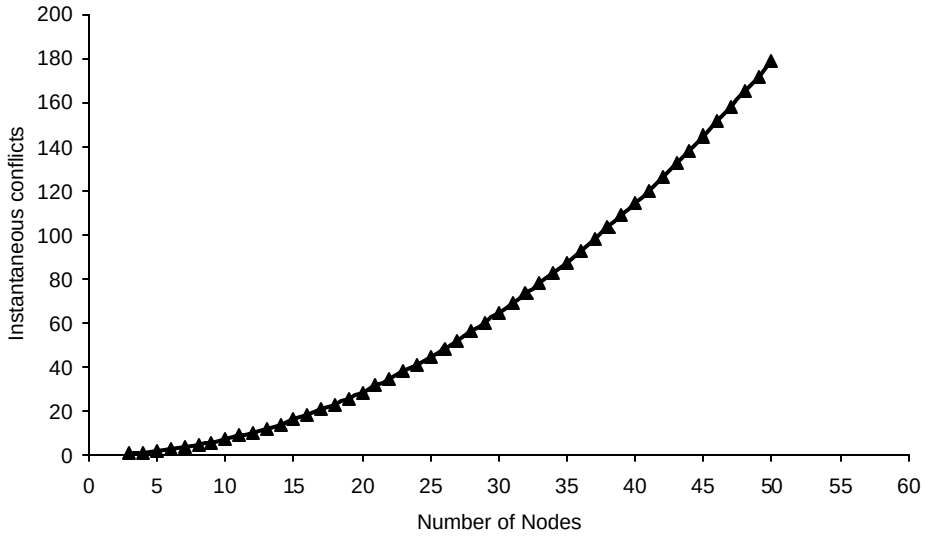
C(n)=
c0n**2.p(W) + c1.n**2.p(UW) + c2.n**2.p(IW) + c3.n**2.p(U) + c4n**2.p(R) + c5.n**2.p(IR)

where 0 < c0, c1, c2, c3, c4 and c5 < 1 and represent the sum of various set of probabilities in the above terms. (In fact, c0=c1=1 in our case)

therefore, C(n) is in theta(n**2)

Queue lengths will follow this n**2 trend which makes it super linear.

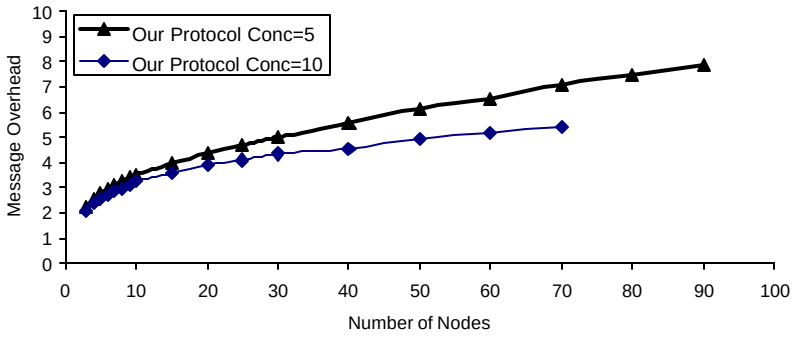
Here is the behavior of the C(n):



This is a new finding that can be studied further in the future.

On the other hand, experiments with various levels of concurrency (C=5 and C=10) indicate that it takes significantly more number of nodes to have a stable behavior of message overhead. In a sense, message overhead behavior loses the clear logarithmic behavior. Response time behavior however, becomes linear but still not logarithmic as expected.

One possible reason for this behavior could be that the tree structure when changed dynamically is more of a list structure than a binary tree structure. Due to that, the propagation path for requests increases linearly (instead of logarithmically) when amortized by number of requests. So the message overhead increases more linearly than logarithmically. The same phenomenon happens to the response time. And with constant degree of concurrency, token transfers should be more dominant than granting.



This problem is crucial for understanding the behavior of the protocol and to make educated progress for priorities too. This may lead to a solution to bound the response time of the requests, at least for soft real-time arena.

4. In the TAO front, after communicating with Bala (Washington University), the framework is clearer. It seems that each client can instantiate its own Concurrency Service object and use the API provided with that. Thus, the protocol can be embedded into those APIs and hidden from the clients. It is still not clear how exactly the concurrency objects will communicate with each other. Probably they can register themselves with the IOR with specific naming conventions and so they can identify each other, get references and invoke some operations on remote

objects, which are the handlers. However, in the ORB net, there might be some of the clients not intending to use the concurrency service. Now, if they are accessing the server object concurrently with other clients, which will interfere with the protocol and mutual exclusion cannot be guaranteed. So, clients need to be aware of the server objects that are concurrent.

Prioritized implementation:

Simulation environment: Everything remains unchanged except the fact that each request has a priority associated with it. The implementation supports N priority levels for N nodes. Here, Each node might have a static priority associated with it or each request can be assigned a priority dynamically which is randomized.
The response time statistics are generated based on:

- Request type: For each type of request the response time is calculated for each level of priority
- Priority level: For each priority level, response times for all types of requests are averaged
- Grand average: Response time is averaged over all priority levels and all request types

Differences with the unprioritized version:

- Format of token transfer message has changed. In unprioritized version, it can be proven that when an incoming request results in a token transfer, the queue at the token node cannot be nonempty. So, the token transfer message did not include the queue transfer. In prioritized version, this property does not hold. So token transfer message includes transferring the local queue from original token node to the new token node, which is piggybacked.
- Queuing and forwarding rule is changed. In unprioritized version, a node can queue an incoming request locally if it satisfies the queue/forward table constraints. In prioritized version, the same table of constraint apply in addition to a constraint that a node cannot queue a request locally if the priority of the pending request is lower than the priority of the incoming request. Queuing such requests may result is priority inversion.
- Freezing mechanism is changed. In unprioritized version, a request cannot be granted if the requested mode is frozen. In prioritized version frozen modes have a level of priority associated with them. When a request with priority p is queued at the token node due to incompatibility, some modes are frozen at the p priority level. Which means that if some other request with priority q > p arrives at the token node and the requested mode is frozen at any priority p < q, then the constraint for the frozen mode is overruled and all the requests with such priority q are granted based on only the compatibility constraints even if the requested modes are frozen. This also means that requests with priority <= p still face the frozen modes constraint as before. However, it may so happen that a request with priority p is queued at the token node and a new request with priority m < p may be granted because the requested mode of the new request is compatible as well as not frozen at priority level >= m.

- New freeze messages are not sent for priority level p when the mode is already frozen at a priority >=p.

- Queue handling is changed. The queue is now ordered based on priority level, usage time and reception time instead of just usage time and reception time. New requests are inserted in-order by performing a linear search.

Next steps

- JPDC submission.
- Deeper investigation of the response time behavior and progress on bounding it based on the priorities.
- Experiment with larger number of nodes on the comparison with Naimi's protocol on Linux cluster.
- Continuing on the TAO front