

Overview

Pramati's Web Container's performance is judged primarily based on its ability to process requests under a heavy load condition with very low latency and high throughput. Often the Web Container is expected to handle a large number of requests without succumbing to its architectural bottlenecks, thus affecting its scalability. Concurrent processing of requests is vital for any web container for staying online in spite of a request flood. Pramati Web Container is built on an innovative threading architecture for reducing the response time and hence boost the performance of the server. Xtreme threading framework achieves faster response times by having finer control on 'degree of concurrency' on resources like CPU, I/O, and Network I/O.

Key Terms

To help you get most of what is discussed in the chapter, we've listed some often used terms in the table below, with their definitions:

Table 1: Terms used

Term	Description
Request	Client trying to invoke a resource from the server
Activity	Processing segment of a request
Thread Pool	A group of threads initialized for concurrent processing
Thread Pool Manager	Manages the activities for the resources
Funneling	An innovative threading framework
Resource	Hardware resource like CPU and Memory
Throughput	Number of requests processed
Logical Pool	A logical pool of threads using a particular resource
Boundless Logical Pool	Threads are allowed to access resources without restrictions
Bounded Logical Pool	Threads are allowed to access resources with restrictions

Thread Pooling Techniques

In this section, we shall examine the various strategies in thread pooling, in the context of middleware requests processing and analyze their relative merits and demerits.

Conventional Thread Pooling

The most straight forward and widely used strategy for request processing is by maintaining a thread pool for a category of requests. By this definition, there would be only one thread pool that would be used internally for implementing the Web Server. Each thread in the thread pool would perform all the tasks involved in request processing. For HTTP request, this would involve:

- 1 Accepting a HTTP request
- 2 Processing the HTTP request
- 3 Reading the data from the file/or executing a servlet to get the data
- 4 Writing the data back to the client

Wherever used, the containers can be configured to control the number of threads in the pool. This is because the optimum number of threads would depend upon the end user environment (type of requests, quality of hardware, network speed etc.)

The disadvantage of this model is that the container doesn't exercise control on the number of threads utilizing various resources simultaneously. So it is possible that all the threads could be utilizing the CPU at one time and could be listening to the open network connections at other time. Whereas to extract maximum benefit, each type of resource would have its own choice of the optimal number of threads operating concurrently. The network is likely to be able to serve 100 simultaneous operations without resulting in higher response times. Whereas it might be sub-optimal to have more than 3 threads utilize the CPU in a particular environment.

Note: Having more threads use the CPU at one time would have the associated costs of context switching, lower throughput and higher response times.

Therefore, if the Server is run with 50 threads, it's possible that all these threads would be contending for processor time simultaneously, thus rendering the Server inefficient.

It is most unlikely to have all threads vying for the same resource type simultaneously. If a single request would take 80% of the total processing time performing network I/O and the remaining 20% of the time utilizing CPU, then if there are 5 threads, on an average 4 threads could be performing I/O and 1 thread performing CPU. However, on the down side, there is likely to be a variance. This variance could be due to the timing of the requests or non-homogeneity of the requests. Also it's not easy to arrive at the 80%-20% figures assumed above in real-time, thus making the choice of optimum number of threads even more difficult.

Thread Pooling Through Dispatcher

To overcome some of the disadvantages of the conventional thread pooling model, a Dispatcher-Worker thread based model can be adapted. In this model, there would be at least two categories of threads. The dispatcher threads would typically bother about obtaining requests from the client, while the worker threads would be performing the actual operation. These two categories of threads may then be separately pooled. Typically the dispatcher threads after performing their operations would place the processed task in a queue. The worker threads would remove the semi-processed

task and carry on further. This Worker-Dispatcher model can be extended to more than one layer to any number of layers.

Application of this model would imply that there could be separate thread pools for various types of activities encountered during HTTP request processing like waiting for new requests, performing I/O operations and CPU intensive processing.

The costs associated with this model are related to the mechanics of transfer of the requests between two categories of threads. First, the operations on the queue which acts as a passage for data between the dispatcher and worker threads, have to be performed in a thread safe manner. This could mean obtaining mutex locks, operating in critical sections which would force thread context switches and avoid JVM optimizations thus resulting in slower overall progress on the requests. Secondly, thread specific data is likely to be attached to each thread and when a new thread is to takeover this semi processed request, the previous threads context needs to be passed to the new thread.

Though this approach is likely to yield better results in some cases, it can't be guaranteed as the benefits obtained out of this approach are linked to the target environment while the costs are fixed. Also, adding extra complexity to the code.

An Insight into the Request Activities

The lifecycle of a request comprises of the following activities:

- 1 Reading header activity
- 2 Request processing activity
- 3 Write response activity
- 4 Waiting for request activity

A typical request processing cycle will start with the waiting for socket state, followed by the new request read state and request processing state. Based on the cache settings, the response is written from the cache or from the file system. Then the response is sent and the socket enters the wait state again. The process and the activity type can be summarized in the following table:

Table 2: Process and Activity type

Process	Activity Type
Waiting for connection	WAIT
Reading request header	Network I/O
Request Processing	CPU
Writing response from the cache	Net I/O
File read	File I/O
Write response	Net I/O
Waiting for connection	WAIT

In a conventional thread modelling setup, where the request is accepted and processed by a thread pool, degree of concurrency varies for each activity state. i.e threads requesting network I/O gets processed faster than the requests for CPU time. This might happen in a scenario where the network bandwidth is high and the processing power is lower. Thus requests tend to wait for CPU processing time more often resulting in a performance bottleneck.

Xtreme Threading

The conventional way of processing request has a performance snag for a system with disparate hardware profile. Hence, it is wise to allocate threads based on the activity performed with a resource. Since the CPU usage is expensive, only few threads are allowed to use the CPU concurrently. All other threads wait for their turn and get processed based on a pre-defined algorithm. This results in response time optimization for larger requests. This process of manipulating the number of threads for processing requests is termed as 'Thread Funneling' or Xtreme threading. The figure below shows the thread optimization using Xtreme threading:

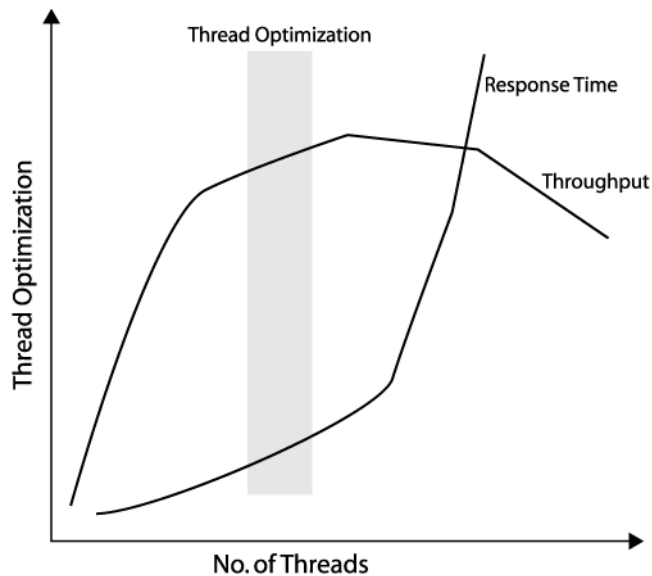


Fig Thread Optimization using Xtreme Threading

If the number of processing threads are more, the response time is less for a period of time and it increases after some time since the threads start waiting for the CPU usage. Also, notice that the throughput of the server remains high for certain period and drops down due to thread congestion. The shaded area shown in the graph provides an optimized threading solution provided by thread funneling technique.

The number of threads utilizing the CPU has an impact on both the throughput of the server and the response time as seen by the clients. At a first glance, it appears as if throughput and response times are inversely related, which is not true. While the throughput measures the efficiency of the code, the response time measures the effectiveness of the server.

Allowing simultaneous access to fewer number of threads in a server could result in suboptimal utilization of CPU (possible as each request perform I/O, DB calls etc. along with utilizing CPU). As the number of threads operating in the server increase, the CPU is likely to be completely utilized. However, some amount of the processing power would be wasted in scheduling the threads, handling the thread context switches etc. As a result, even though the CPU could be entirely utilized, the server process may not get that entire power and there would be wastage. Any further increase in the number of threads, beyond the 100% CPU utilization stage, would only add to the further overhead resulting in higher wastage of processing power and lesser availability of hardware resources to the server.

Allowing access to fewer number of threads would result in best response time. With the increase in number of overall threads (without any fine grained control on these threads), the response times would increase. This happens till the Throughput reaches it's maximum, this could only mean that the response is faster but processing power is not being fully utilized.

As can be seen from the graph, response time deteriorates at a much faster rate once the throughput reaches the maximum. Hence it is important to establish a band on the number of threads so that the server is very efficient. Xtreme threading is an attempt to eliminate the variance, and to make sure that the server always operates in the optimal band.

Xtreme Threading Architecture

Xtreme Threading is an approach that would enable choosing the right thread processing model based upon the target environment and the type of request processing. Instead of forcing a decision on the choice of Thread Pooling model at the time of product development, this model is expected to postpone the decision to the server runtime.

In this model, the request processing cycle would be divided into various processing portions depending primarily on the type of resource being used (like CPU, network I/O, file I/O, or wait for request). Each processing portion would implement an interface that accepts the previous Task state and returns the current Task state. The current task state would then be passed to the subsequent processing portion.

```
<thread-funnel worker_thread_count = "50" enabled="false">
  <!-- <activity-type name="accept_requests" num="1"/> -->
  <activity-type name="request_processing" num="10" />
  <activity-type name="file_io" num="30" />
  <activity-type name="network_io" num="30" />
  <activity-type name="keepalive_waits" num="40" />
  <service_unavailable_threshold num_waiting="20" />
</thread-funnel>
```

Activities

Each processing portion for a resource will have an optimal number of concurrent threads above which performance will deteriorate due to the cost of thread management (swapping threads back and forth). In processing portions of WAIT, network I/O latency of resource is so high that it does not really matter to have many concurrent threads executing.

In Pramati Thread Funneling model, these processing portions are termed as “Activities”. And activities that use particular kind of resource are grouped into a logical pool. Threads entering the processing portion/Activity in their execution path obtain license from their corresponding logical pool. The Logical pool implementations can implement many policies that process the license request by a thread.

Each activity is classified on the resource that is used (CPU, IO, Wait) and the type of processing done on the request. During the processing of a request, there could be various intervals where CPU could be used, separated by intervals where database /IO work is done. Each interval should be then marked as an activity. And all the activities that use CPU can be linked to each other by mapping them to the Logical Pool, which identifies the CPU resource. Activities are associated with instance of Logical Pool generally termed as HW Resource Manager. The HW Resource Manager maintains the number of threads waiting on a resource.

Logical Pool

A set of activities are mapped to an instance of logical pool. Logical pools can be written to implement a particular resource policy. Typically one resource is expected to be attached with one logical pool. There is expected to be a n-to-1 relationship between activities and logical pools. This initialization should be done by the users of this package.

Gate Keeper

Users of the thread funneling framework will provide information on types of activities in the execution path of the request processing code. The Gate Keeper class will then get license to serve the activity from the logical pool associated.

Priority Processing

Depending on the configuration, the thread funneling framework will validate requests being served by the container and assign priorities to the thread serving the request. Higher priority threads get preference in scheduling while they switch their states at all marked points.

```
<priority-processing enabled = "false">
<application name="TestpriorityWeb" virtual_host="default">
  <priority-groups realm="system" level="8">
    <groups>administrator, everybody</groups>
    <users> root, test </users>  <!-- optional.. to be impl later -->
  </priority-groups>
```

```
</application>

<application name="admin" virtual_host="default">
  <priority-groups realm="system" level="8">
    <!--<groups>administrator, everybody</groups>          -->
  </priority-groups>
  <users> root </users>
</application>
</priority-processing>
```

Thread Pool Manager

Xtreme Threading Framework has a Thread Pool Manager, which is aware of the various processing portions of a resource (obtained through configuration or registration). This Thread Pool Manager can be started with a configuration that would specify the relation between the various processing portions and corresponding logical thread pools. One configuration could be where a separate thread pool is specified for each processing portion (Multi dispatcher model). At the other extreme there could be a single thread pool for all processing portions (Traditional Thread Pooling model). In this case the container need not incur cost of passing the request through queues. It's also possible to have a variety of other configurations taking the middle ground.

This extreme threading approach would thus enable to chose the optimal configuration based upon the target environment and by incurring only as much extra cost as is required. A truly extreme threading approach would involve letting the Thread Pool Manager dynamically determine the number of Thread Pools and their respective sizes intelligently based upon the target environment, with the knowledge of costs to be incurred in various configurations. Such an approach can even let the administrator know the amount of improvement achieved by utilizing this extreme threading approach as compared to others.

Summary

This chapter discussed how to increase the performance of Pramati Web Container. We discussed a number of thread pooling techniques and also saw what happens when a HTTP request is received by the Web container. Xtreme Threading is a process by which the performance of the Web container can improve the response time. This is done by way of dividing the request processing cycle into various portions depending on the type of resources. Logical pools are maintained for each activity, and the pool size is configurable. Finally we discussed what priority processing is and the role of a Thread Pool Manager.

