**Modules:**

* Request Handler
* Resource Controllers
* Resource Aware Scheduler
* Thread Pool
* Thread pool performance monitor

1. **Request Handler:**

Many server applications, such as Web servers, database servers, file servers, or mail servers, are oriented around processing a large number of short tasks that arrive from some remote source. A request arrives at the server in some manner, which might be through a network protocol (such as HTTP, FTP, or POP), through a JMS queue, or perhaps by polling a database. Regardless of how the request arrives, it is often the case in server applications that the processing of each individual task is short-lived and the number of requests is large.

One simplistic model for building a server application would be to create a new thread each time a request arrives and service the request in the new thread. This approach actually works fine for prototyping. Normally active threads consume system resources.

The thread-per-task approach works quite well with a small number of long-running tasks. The single-background-thread approach works quite well so long as scheduling predictability is not important, as is the case with low-priority background tasks. However, most server applications are oriented around processing large numbers of short-lived tasks or subtasks, and it is desirable to have a mechanism for efficiently processing these tasks with low overhead, as well as some measure of resource management and timing predictability.

1. **Resource Controllers:**

The controller consists of several components. A *monitor* measures response times for each request passing in to the Thread Pool. Requests are tagged with the current time when they enter the service. At each pool, the request’s response time is calculated as the time it leaves S minus the time it entered the system. While this approach does not measure network effects, we expect that under overload the greatest contributor to perceived request latency will be intra-service response time.

The measured 90th-percentile response time over some interval is passed to the *controller* that adjusts the *admission control parameters* based on the administrator-supplied response-time *target*. In the current design, the controller adjusts the rate at which new requests are admitted into the thread pool or the request are admitted into the next thread pool.

1. **Resource Aware Scheduler:**

A resource handler is one that receives requests from request handler and redirects it to the thread pools to get the job done. But it is not as simple as it looks. The main function of resource handler is, it has to redirect the request, which it got from the resource handler to the thread pool where there is no overloading. This can be achieved through constant listening of thread pool performance monitor. When a request arrives it will look for thread pool performance monitor to analyze the load details for the particular instance. Based on the knowledge gained it will then redirect to the thread pool where the load is minimum. In its worse case if it cannot find any thread pools to be free, the incoming requests will be stored in a buffer memory.

SSA provides application-specific scheduling for thread-based applications. Since SSA uses a cooperative threading model, we can view an application as a sequence of stages, where the stages are separated by blocking points. Our methods are more powerful, however, in that they deduce the stages automatically and have direct knowledge of the resources used by each stage, thus enabling finer-grained dynamic scheduling decisions. In particular, we use this automated scheduling to provide admission control and to improve response time. Our approach allows SSA to provide sophisticated, application-specific scheduling without requiring the programmer to use complex or brittle tuning APIs. Thus, we can improve performance and scalability without compromising the simplicity of the threaded programming model.

Most existing event systems prioritize event handlers statically.. SSA goes one step further by introducing the notion of resource-aware scheduling. In this section, we show how to use resource-aware scheduling that is both transparent and application-specific. Our strategy for resource-aware scheduling has three parts:

1. Keep track of resource utilization levels and decide dynamically if each resource is at its limit.

2. Annotate each node with the resources used on its outgoing edges so we can predict the impact on each resource should we schedule threads from that node.

3. Dynamically prioritize nodes (and thus threads) for scheduling based on Information from the first two parts.

For each resource, we increase utilization until it reaches maximum capacity (so long as we don’t overload another resource), and then we throttle back by scheduling nodes that release that resource. When resource usage is low, we want to preferentially schedule nodes that consume that resource, under the assumption that doing so will increase throughput. More importantly, when a resource is overbooked, we preferentially schedule nodes that release the resource to avoid thrashing. This combination, when used with some hysteresis, tends to keep the system at full throttle without the risk of thrashing. Additionally, resource-aware scheduling provides a natural, workload-sensitive form of admission control, since tasks near completion tend to release resources, whereas new tasks allocate them. This strategy is completely adaptive, in that the scheduler responds to changes resource consumption due to both the type of work being done and offered load.

1. **Thread Pool:**

* **Risks of using thread pools:**

While the thread pool is a powerful mechanism for structuring multithreaded applications, it is not without risk. Applications built with thread pools are subject to all the same concurrency risks as any other multithreaded application, such as synchronization errors and deadlock, and a few other risks specific to thread pools as well, such as pool-related deadlock, resource thrashing, and thread leakage.

* Deadlock:

With any multithreaded application, there is a risk of deadlock. A set of processes or threads is said to be *deadlocked* when each is waiting for an event that only another process in the set can cause. While deadlock is a risk in any multithreaded program, thread pools introduce another opportunity for deadlock, where all pool threads are executing tasks that are blocked waiting for the results of another task on the queue, but the other task cannot run because there is no unoccupied thread available. This can happen when thread pools are used to implement simulations involving many interacting objects, and the simulated objects can send queries to one another that then execute as queued tasks, and the querying object waits synchronously for the response.

* Resource thrashing:

Threads consume numerous resources, including memory and other system resources. Besides the memory required for the Thread object, each thread requires two execution call stacks, which can be large. In addition, the JVM will likely create a native thread for each Java thread, which will consume additional system resources. Finally, while the scheduling overhead of switching between threads is small, with many threads context switching can become a significant drag on your program's performance.

* Concurrency errors:

Thread pools and other queuing mechanisms rely on the use of wait() and notify() methods, which can be tricky. If coded incorrectly, it is possible for notifications to be lost, resulting in threads remaining in an idle state even though there is work in the queue to be processed.

* Thread leakage:

A significant risk in all kinds of thread pools is thread leakage, which occurs when a thread is removed from the pool to perform a task, but is not returned to the pool when the task completes. One way this happens is when the task throws a Runtime Exception or an Error. If the pool class does not catch these, then the thread will simply exit and the size of the thread pool will be permanently reduced by one. When this happens enough times, the thread pool will eventually be empty, and the system will stall because no threads are available to process tasks.

Tasks that permanently stall, such as those that potentially wait forever for resources that are not guaranteed to become available or for input from users who may have gone home, can also cause the equivalent of thread leakage. If a thread is permanently consumed with such a task, it has effectively been removed from the pool. Such tasks should either be given their own thread or wait only for a limited time.

* Request overload:

It is possible for a server to simply be overwhelmed with requests. In this case, we may not want to queue every incoming request to our work queue, because the tasks queued for execution may consume too many system resources and cause resource starvation. It is up to you to decide what to do in this case; in some situations, you may be able to simply throw the request away, relying on higher-level protocols to retry the request later, or you may want to refuse the request with a response indicating that the server is temporarily busy.

**5. Thread pool performance monitor:**

It looks after the performance of the thread pools associated with it through a response time controller “T**”**. The response time controller will monitor the load of the Input and Output of its own thread pool and send information to the Thread Pool Performance Monitor whenever there is a fluctuation in performance. The fluctuations in load will be calculated by monitoring the number of requests given as input to the number of requests processed at the output for any unit time. The thread pool performance monitor in-turn sends information to help the resource scheduler in directing the request to the appropriate thread pool for better performance and resource utilization. A separate table will be maintained in the thread pool performance monitor to update the load fluctuations of the thread pool associated with it. This monitoring system will send a copy of the table to the resource scheduler whenever there is an update.