Scalability Design Patterns

Kanwardeep Singh Ahluwalia
81-A, Punjabi Bagh,
Patiala 147001
India
kanwardeep@gmail.com
+91 98110 16337

Abstract

Achieving highest possible scalability is a complex combination of many factors. This paper presents a pattern language that can be used to make a system highly scalable.

Introduction

Scalability is a desirable property of a system, a network, or a process, which indicates its ability to either handle growing amounts of work in a graceful manner, or to be readily enlarged. For example, it can refer to the capability of a system to increase total throughput under an increased load when resources (may be hardware) are added.

There are various ways to make a system scalable – some of these ways being driven by the domain to which the system belongs. For example, scientific computing related system may stress on faster algorithms. Similarly, a system involving complex calculation may stress more on the faster hardware. While a web based application may just opt for a cluster of low or medium end machines.

Scalability has been an area of concern for many years. The approach to achieve scalability has also been changing. More affordable and abundant hardware has made possible to focus on adding hardware as one of the simplest solutions to enhance the scalability of a system. While two decades ago, the focus used to be more on the faster algorithms, so as to save the cost of buying highly priced hardware and hence do more processing in little time.

There have also been different perceptions on how to measure scalability. In case of a transaction oriented system, some people would dictate scalability requirements in terms of maximum number of simultaneous users supported by the system. While others would prefer to dictate the scalability requirement in terms of the maximum transactions processed per unit of time by the system. For example, the requirement could say that the system initially should be able to support 50 simultaneous users, but it should be scalable enough to support up to 500 simultaneous users.

Figure 1 describes a way of measuring the scalability of a system; a graph is plotted with load (in terms of simultaneous) on the X-axis and throughput (in terms of transactions per unit of time) on the Y-axis. As shown in the figure, the throughput of the system usually increases initially as the system is exposed to an increasing load. After a certain limit the throughput of the system remains constant for a while, even when subjected to an increased load. However, if the load keeps on increasing, then a point comes after which the throughput of the system starts decreasing. This point is usually known as "knee point". This may happen due to many reasons, like the system may be feeling the scarcity of hardware resources, or the system may be experiencing a lot of locking issues, due to which the overhead increases and hence the net throughput decreases. The system may even fail if the load is increased beyond 'A'. Hence, the system is scalable enough only to support 'A' simultaneous users.

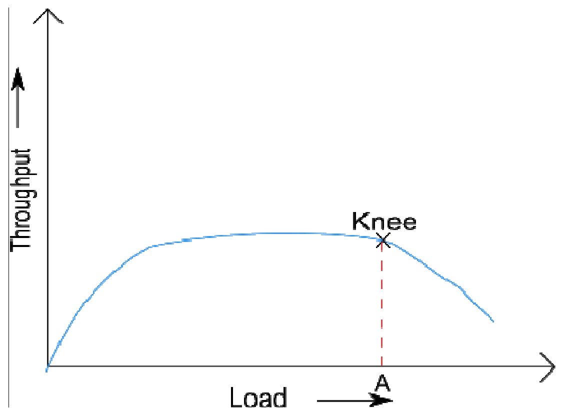

Figure 1: Measuring Scalability

Figure 2 shows the system behavior under increasing load after it has been made more 'scalable' by following one of the approaches defined in this paper. For instance, it can be as simple as adding some more processors to the existing hardware. The following figure shows that the system has become more scalable as it is now able to handle more input load. The 'knee point' A' has drifted further away.

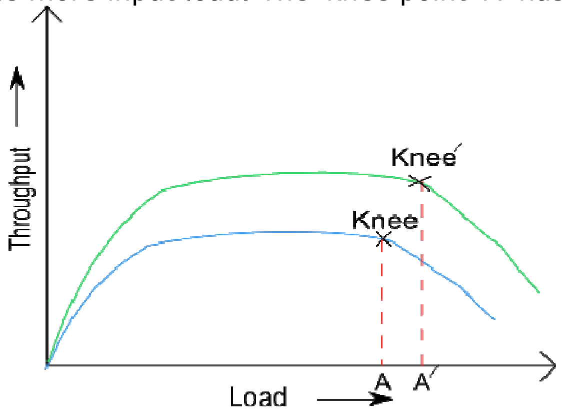

Figure 2: Comparing Scalability

It is also noted that there is a general perception that a system can scale proportional to the hardware added to the system. This perception might not be generally true as has been proved by Amdahl's Law. For example: suppose a portion of a program can be sped up by parallelizing it. Suppose we can improve 70% of a module by parallelizing it, and run on four CPUs instead of one. If α is the fraction of a calculation that is sequential, and 1 − α is the fraction that can be parallelized, then the maximum speedup that can be achieved by using P processors is given according to Amdahl's Law: $\frac{1}{\alpha + \frac{1-\alpha}{P}}$. Substituting the values for this example, we get $\frac{1}{0.3 + \frac{1-0.3}{4}} = 2.105$. If we double the compute power to 8 processors we get $\frac{1}{0.3 + \frac{1-0.3}{8}} = 2.581$. Doubling the processing power has only improved the speedup by roughly one-fifth. If the whole problem was parallelizable, we would, of course, expect the speed up to double also. Therefore, throwing in more hardware is not necessarily the optimal approach.

The patterns in this paper address the architectural and design choices that one must consider while designing a scalable system. These patterns are best suited for a transaction oriented systems. These patterns are not discussing the programming techniques that can be used to implement these patterns. The intended audience includes system architects and designers who are designing scalable systems.

Language Map

Figure 3 presents the Scalability patterns language map.

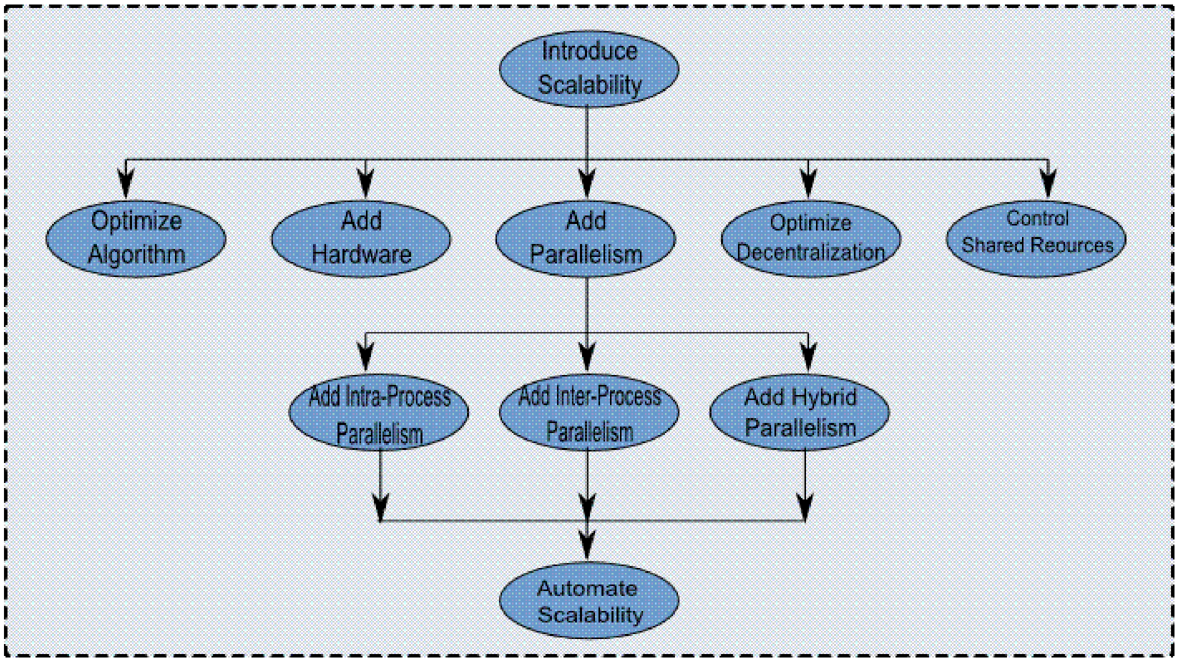

Figure 3: Scalability Patterns Language Map

Pattern 1: Introduce Scalability

Context:

System that does not want to deteriorate its performance when the load increases.

Problem:

How can a system maintain its performance with increasing input load?

Forces:

- System should perform faster so as to do more processing in lesser time, i.e., able to maintain its processing rate when load is increased.
- System should make full use of the available resources.

Solution:

The system has to be scalable in order to handle increased load. This can be achieved by any or all of the following.

“Optimize Algorithm”

“Add Hardware”

“Introduce Parallelism”

“Optimize Decentralization”

“Control Shared Resources”

Resulting Context:

The system is able to maintain its throughput with the increased load, as well as optimized usage of system resources is ensured.

Known Uses:

Application servers like Websphere and Weblogic are popular for their in-built scalability features, like thread pool for parallel processing, cluster enabled configuration, ability to scale with additional hardware.

Related Patterns:

Optimize Algorithm

Add Hardware

Introduce Parallelism

Optimize Decentralization

Control Shared Resources

Pattern 2: Optimize Algorithm

Context:

System that wants to increase its scalability without adding hardware or parallel processing to the system.

Problem:

How can a system enhance its throughout with without adding new hardware resources to the system?

Forces:

- System should be able increase its performance without compromising on the functionality.
- System should be able to maintain its transaction processing rate.
- System should make full use of the available resources.

Solution:

The key to the solution is to identify the areas those can be optimized for performance when the input load increases. The aim is to identify tasks which can be completed in a shorter period to save processing time. This shall result in overall throughput improvement of the system by allowing the saved CPU time to be allocated for growing work.

These areas can in some cases be identified by carrying a code walkthrough to identify areas where smarter algorithms can help improving the performance.

If code walkthrough does not help, then certain tools can be used to identify the areas which are consuming most of the time during processing. These tools (like IBM Rational Quantifier) help identifying the areas (functions) which are eating up most of the CPU time.

There can be various ways to reduce the processing time of the identified heavy areas. This may mean to use an alternate algorithm which shall result in keeping the end result same but complete the same task in a shorter duration.

Resulting Context:

The system is able to maintain its throughput with the increased load, as well as optimized usage of system resources is ensured.

Known Uses:

Running time of Insertion sort is O (n*n), while that of Quick sort is O(n lg n). Hence, quick sort scales better with n as compared to insertion sort as depicted in the Figure 4.

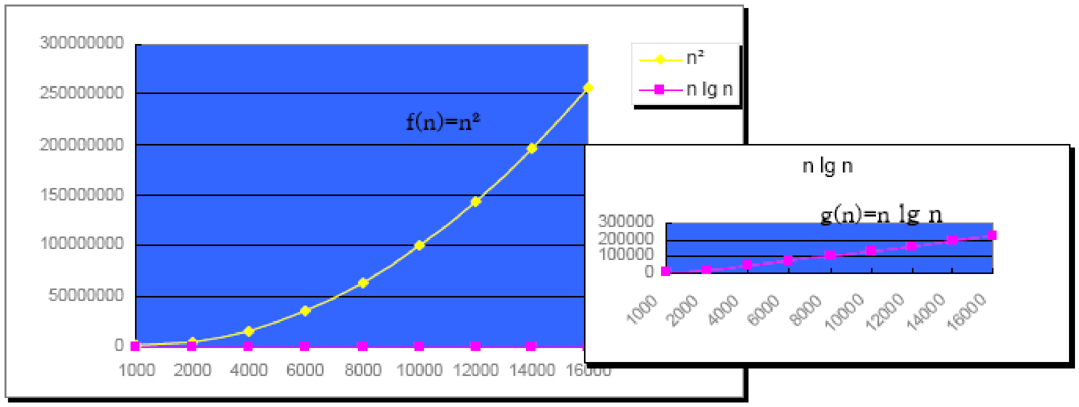

Figure 4 : Comparison between the function of n² and n lg n

Another very well known use of enhancing algorithms is by using 'join' statement in the SQL statements rather than having inner query. 'Inner' query is less scalable as compared to the 'join' query with growing data.

Related Patterns:

Introduce Scalability

Pattern 3: Add Hardware

Context:

System that uses various hardware and wants to increase its scalability and has access to the additional hardware.

Problem:

How can a system entertain increased load without changing the code?

Forces:

- It should be possible to identify the scarce hardware resources.
- System should be able to maintain its transaction processing rate with the increased load.

Solution:

The key to the solution is to identify the hardware resources which are becoming scarce for the system.

The scarce hardware resources can be identified by using resource monitoring tools (like prstat/top).

Once the scarce hardware resources are identified, the next step is to add them in efficient quantity.

The hardware resources can be added to the same existing physical hardware or it may be added as a new physical hardware.

Adding hardware to the existing physical node will result in what is commonly known as "vertical Scalability", where as, adding hardware as a separate new node will result in enhanced "horizontal scalability".

Resulting Context:

The system is able to maintain its throughput with the increased load.

Known Uses:

Adding RAM to the existing machine incase it is detected that the system requires more memory.

Adding a separate machine to a cluster in case it is found that all the existing machines in the cluster are being utilized to their full capacity.

Related Patterns:

Introduce Scalability

Pattern 4: Introduce Parallelism

Context:

System that does not want to deteriorate its performance when the load increases and has the capability to split the work in to pieces which can be completed simultaneously.

Problem:

How can a system maintain its performance with increasing input load?

Forces:

- System should be able to do process multiple tasks at the same time.
- System should be able to maintain its transaction processing rate with increasing input.
- System should make full use of the available resources.
- System should be able to decide the level of parallelism to be introduced versus the complication added by parallelism.

Solution:

The key to a scalable design is to process multiple transactions in the system simultaneously. The system should do parallel processing in order to maintain the throughput with the increasing load.

The work should be divided in to pieces that can be done simultaneously so as to do more processing in the same time. Longer the task is identified for parallel processing, the more will it add to the scalability of the system, as it would considerably increase the throughput of the system enabling it to handle much higher loads. Parallelism will also ensure optimized usage of system resources such a CPU.

Further parallelism can be introduced in a way to process multiple transactions simultaneously as shown in the Figure 1 or it can also be introduced to process various tasks in a single transaction simultaneously as shown in the Figure 6. In the first case, each parallel processing unit is replica of the other as all such units would be processing similar transactions simultaneously. However, in the later case, each parallel would be a specialized one and may not be similar to the others.

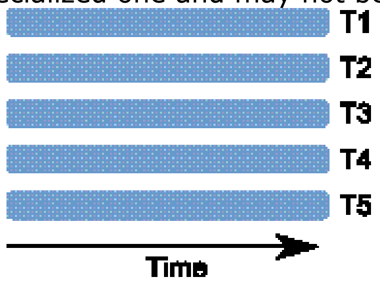

Figure 5 : Multiple Simultaneous Transactions

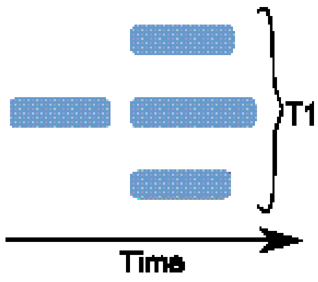

Figure 6 : Single transaction split in to multiple parallel tasks

It is not always very easy to introduce parallelism in an existing system. Hence, parallelism is something which should be considered right at the time of designing a scalable system.

The down side of adding parallelism is that it usually makes a system complex to maintain, debug, and a bad design will make it more prone to the defects and can result in losing the integrity of data as well. Hence, it is usually not the first choice of many designers to enhance the scalability of the system.

Parallelism can come in different forms as described briefly below.

- A system can have multiple threads (Intra-process scalability)
- Or, it can have multiple processes (Inter-process scalability)
- Or, a system can have mix of both of the above (Hybrid scalability) in order to process multiple transactions simultaneously.

Resulting Context:

The system is able to maintain its throughput with the increased load, as well as optimized usage of system resources is ensured.

Known Uses:

One of the common examples is the J2EE server architecture, where in HTTP requests are processed simultaneously by worker threads as shown in Figure 7.

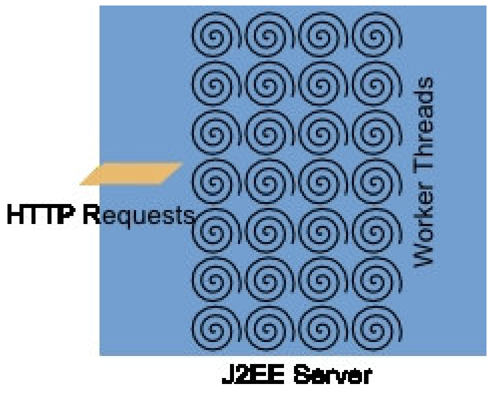

Figure 7 : Worker threads in an application server

Another well known example is Cluster computing, where in multiple nodes in the cluster are processing transactions simultaneously as shown in the Figure 8.

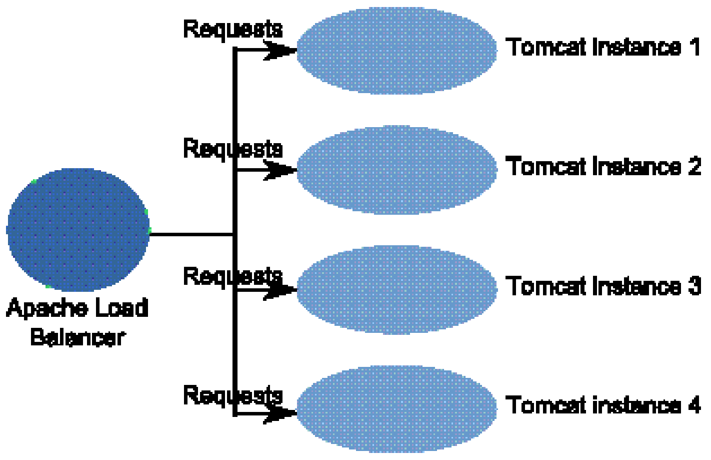

Figure 8 : Apache - Tomcat Cluster

Another known use of splitting a single transaction in to multiple parallel tasks can be found in many applications where one thread in busy performing disk access while the other thread is busy performing the network access. Both these tasks (disk and network access) being part of the same transaction.

Related Patterns:

Introduce Scalability
Intra-process Parallelism
Inter-process Parallelism
Hybrid Parallelism

Pattern 5: Optimize Decentralization

Context:

System that has introduced parallelism, but is not able to scale due to bottlenecks, (the areas related to resources those are required by parallel processing paths.)

Problem:

How can a system do true parallel processing by avoiding the bottlenecks?

Forces:

- Centralization can be a bottleneck
- More bottlenecks the system has, less scalable it is
- It's hard to avoid centralization.

Solution:

The system should be designed to avoid bottlenecks. All such bottlenecks should be avoided by following decentralized approach, where in processing is not dependent on a particular resource, instead multiple resources are provided to make each parallel path independent enough not to be burden or dependent on the other paths.

Various tools can be used to identify scaling bottlenecks like IBM Rational Quantify, which lists the graphical display of the request flow. However, it is not easy to remove bottlenecks. But the most common techniques to remove the scaling bottlenecks involve providing individual copy of the resource that is creating bottleneck to each contender. Others include minimizing the bottleneck area, so that the impact of bottleneck is reduced.

However, practically it is not possible to completely get rid of centralized processing; Instead, it may be beneficial at some places. In addition, predicting bottlenecks is hard, but measuring them is easy. Therefore, an incremental approach is suggested to optimize decentralization, where in a bottleneck detected during processing should be removed and then the system should be again observed for further bottlenecks. This shall ensure that only centralized processing areas those are posing as bottlenecks are removed.

Resulting Context:

The system is able to do parallel processing without running in to scaling bottlenecks.

Known Uses:

Apache-Tomcat cluster having multiple Tomcat nodes – each one of them processing HTTP requests independently. However, these different instances may be talking to the same database instance.

Related Patterns:

Introduce Scalability

Pattern 6: Control Shared Resources

Context:

System that has introduced parallelism and wants to access some of the shared resources while doing parallel processing.

Problem:

How should system be able to do parallel processing when there are some resources to be shared across parallel paths?

Forces:

- System should be able to share the resources so as to execute tasks in parallel.
- System should avoid race conditions and shared resources should not get corrupted.

Solution:

The system should identify inevitable shared resources in the system. These shared resources should be further categorized in to "Access Only" and "Modifiable" resources. "Access Only" resources should not be a problem while they are accessed by different nodes during parallel processing. Special care has to be taken for "Modifiable" shared resources to maintain their integrity while they are being modified by multiple nodes simultaneously.

Special care has to be taken to prevent the corruption of a shared resource in a scenario where one parallel processing unit is trying to read a "Modifiable" resource and while another parallel processing unit is trying to modify the same resource. The most common solution to prevent the corruption of shared resources is to capture a lock on the shared resource, modify it and release the lock. This shall ensure that the shared resource is being modified by only one parallel processing unit at a time.

Using locks is not always as trivial as it sounds. Locks should be carefully used so as to avoid any deadlock scenario which may occur due to following conditions.

- A parallel processing already holding lock requests the same lock again.
- Two or more parallel processing units form a circular chain where each one waits for a lock that the next one in the chain holds.

These deadlocks are hard to detect. An in depth code walkthrough or some tools like Sun Studio Thread Analyzer can be used to detect thread locks.

Resulting Context:

The system is able to access its shared resources without corrupting them while doing parallel processing.

Known Uses:

A global data structure to be modified by the multiple threads in the system is usually accessed via a Mutex lock.

Related Patterns:

Introduce Scalability

Pattern 7: Add Intra-process parallelism

Context:

System that wants to scale within a single process and has the capability to create multiple threads.

Problem:

How can a system be able to scale when there is only one process available to handle the entire input load?

Forces:

- Single process should be able to exploit parallelism to handle the increased load.
- The single process should make optimized usage of hardware resources to handle the increased load.
- System should be able to decide the level of parallelism to be introduced versus the complication added by parallelism.

Solution:

The system should make use of threads to do the parallel processing. For most of the existing operating systems, these threads would compete for the hardware resources allocation in the same way as processes would do. Multi-threading will allow the process to make use of multiple cores, CPUs and/or hyperthreading. This will ensure that even through there is a single process in the system; it is able to increase its proportionate usage of hardware resources to handle increases input load.

Multi-threading will also allow the system overlap various tasks independent of each other like one thread can communicate with the disk while the other thread at the same time can communicate with the network. Multi-threading also allows to overlap some CPU intensive task with the non-CPU intensive task, for instance, one thread could be doing receiving data from the network while the other one could be doing parsing of the already received data.

Figure 9 shows a process with a single thread on the left hand side of the dotted line accessing only one CPU at a time; where as the process with Intra-process parallelism is shown on the right hand side of the dotted line having multiple threads accessing more than one CPUs at a time.

The number of threads in a process can be part of a thread pool which has configurable number of threads depending on the level of scalability requirements.

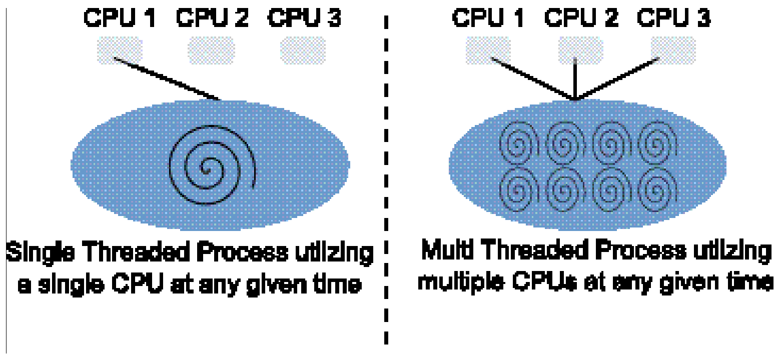
CPU 1
CPU 2
CPU 3
CPU 1
CPU 2
CPU 3
Single Threaded Process utilizing a single CPU at any given time
Multi Threaded Process utilizing multiple CPUs at any given time

Figure 9 : Display of Intra Process Parallelism

Threads add a lot of complexity to the system. It is difficult to maintain and debug a multi-threaded system. Hence, they should be used with care and be used only if required. Also, adding very large number of threads in a process may not result in an increased scalability, as the overhead of locking and synchronization to access shared resources may surpass the benefit of adding more threads to the system after a certain point.

Resulting Context:

The system is able to do parallel processing even from with in a single process by making use of threads.

Known Uses:

A servlet container like Tomcat makes use of multiple threads to bring scalability.

Related Patterns:

Intra-process auto scalable systems

Pattern 8: Add Inter-process parallelism

Context:

System that can not scale using ”Intra-process parallelism” and has the capability to spawn multiple processes.

Problem:

How will a system handle the increased load when it can not spawn multiple threads with in a process?

Forces:

- The system should make optimized usage of hardware resources to handle the increased load.
- The system needs to have collaboration between various processes.
- System should be able to decide the level of parallelism to be introduced versus the complication added by parallelism.

Solution:

The system needs to replicate its process by spawning their multiple instances. All these multiple instances need to coordinate with each other to handle the load in a distributed manner. These processes can coordinate with each other with the help of a load balancer, which helps in assigning the task to each process.

The structure is shown below in the Figure 10. The picture on the left hand side of the dotted line shows the there is a single process to handle the entire load sent by client. While the picture on the right hand side shows that the system has multiple processes to handle the load. There is a load balancer to load the multiple processes in the system. The client is not aware of the multiple processes in the system, but definitely enjoys the enhanced scalability.

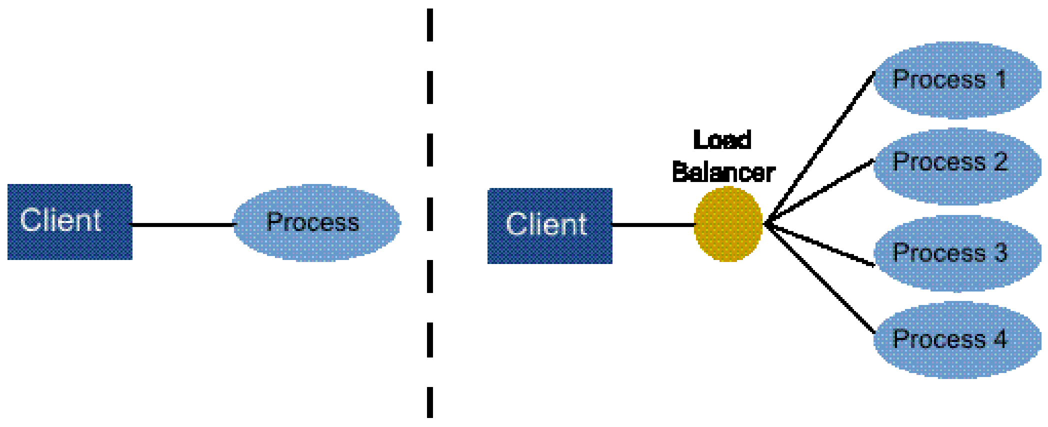

Figure 10 : Effect of introducing Inter Process Parallelism

Each process instance can handle each transaction independently or each transaction can be simultaneously processed by multiple processes. In case, each process is able to handle transactions independently, then each process is replica of the other. However, in case each transaction has to be processed by multiple processes, then each process is a specialized process providing a specific functionality.

Having too many processes may not actually make the system more scalable. This may happen because the overhead introduced due to coordination between the processes may takeover the gain from having multiple processes. Hence, the system should only introduce optimum number of processes. This optimum number can be determined by increasing the number of processes gradually and then observing the gain in the scalability.

Resulting Context:

The system is now able to handle the increased load by spawning its multiple processes without compromising on the throughput.

Known Uses:

Apache-Tomcat cluster is a classic example of having multiple identical Tomcat processes working together to provide inter-process parallelism.

Another example of a transaction being processed by multiple clients comes from a centralized logger used in most of the enterprise applications as shown in Figure 11. Logging required by all the transactions is provided by a centralized process which talks to other processes through asynchronous message queue like JMS. Here the main processes are delegating the logging to a centralized logger without waiting for actual logging to happen, these processes move on further to process the remaining transaction.

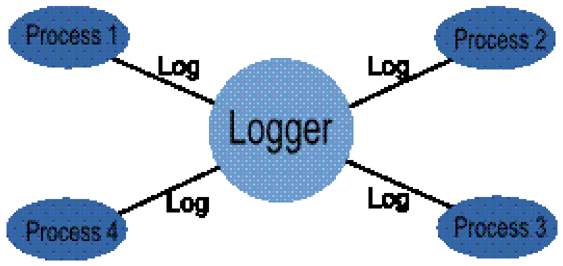

Figure 11 : Centralized Logger - An example of a transaction being process by multiple processes

Related Patterns:

Inter-process auto scalable systems

Pattern 9: Add Hybrid parallelism

Context:

System that has the capability to spawn both multiple threads as well as processes.

Problem:

How should a system handle the increased load when it can spawn both threads and processes?

Forces:

-
- System should be able to decide the level of parallelism to be introduced by multiple threads as well as processes versus the complication added by each of these parallelisms.
- System can gain by having multi-threaded processes.
- There is a limit to which a process can handle the increased load by adding threads, as after a certain point, the overhead of concurrency takes over the benefit of parallelism provided by multi-threading.
- System can gain by having multiple processes.
- There is a limit to which a system can gain by increasing number of processes.

Solution:

The System needs to spawn multiple process instances as specified in the pattern "Inter-process Parallelism" after a process is no more able to scale by increasing its number of threads as specified in the pattern "Intra-process Parallelism". This has been depicted in Figure 12.

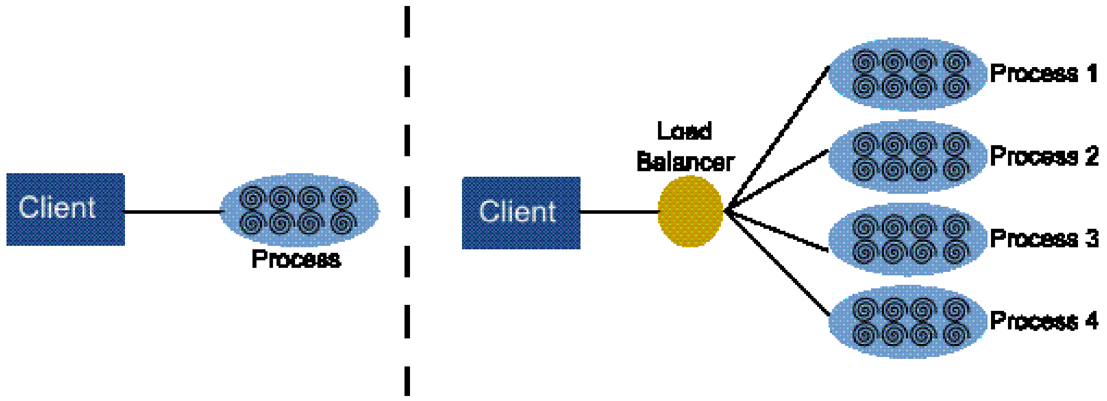

Figure 12 : Hybrid Parallelism

The number of processes can increase depending up on the input load. All these multiple instances need to coordinate with each other to handle the load in a distributed manner. This coordination can happen with the help of a Load balancer.

Resulting Context:

The system is now able to handle the increased load by following a hybrid approach both by increasing number of threads and number of processes without compromising on the throughput.

Known Uses:

In a typical J2EE based application cluster, there are number of application container processes. Each of these processes has in turn number of threads for simultaneous load processing.

Related Patterns:

Hybrid auto scalable systems

Pattern 10: Automate Scalability

Context:

System using Intra, Inter or Hybrid Parallelism with a lot of varying and unpredictable load.

Problem:

How can the system automatically scale up or down to handle the increased or decreased load?

Forces:

- System should be able to detect that with the given configuration it is not possible to handle the increased load.
- System should be able to automatically define the amount (number of threads and processes) by which it has to scale.
- The automation may add to the complexity in the system. Hence, the benefit of automation should be weighed against the complexity added by the automation.

Solution:

The system needs to have a monitoring entity that measures the current throughput and has the ability to increase or decrease the number of threads or processes in the system.

As soon as the monitoring entity detects that the transactions rate is dropping and nearing the minimum pre-configured throughput, then it should gradually increase the number of threads in the thread pool (in case of Intra or Hybrid parallel system) or number of processes (in case of Inter or Hybrid parallel system). This should pump up the transaction rate. The number of threads or processes should be stopped from increasing once the system increases its throughput to the desired rate.

Additionally, the monitoring entity should decrease the number of processes or threads once it observes that the input load is decreasing. This can be done by having threads with an idle time-out period after which they should die; if the load on the system decreases. Similarly, the monitor may decide to kill the processes in excess in the Last in First out (LIFO) manner.

Resulting Context:

The system is able to dynamically adjust its number of threads and processes in order to handle the increased load with the same throughout.

Known Uses:

In a typical J2EE based application cluster, there are number of application container processes. Each of these processes has in turn number of threads for simultaneous load processing.

Related Patterns:

Intra Process Parallelism

Inter Process Parallelism

Hybrid Parallelism

Acknowledgements

I would like to thank Berna Massingill for her feedback and encouragement during shepherding of these patterns. She has been very kind to provide in-depth suggestions and comments on these patterns.

References

[1] Apache Tomcat Cluster Description at http://tomcat.apache.org/tomcat-5.0-doc/cluster-howto.html
[2] http://en.wikipedia.org/wiki/Scalability
[3] Sort Comparisons at www-users.cs.umn.edu/~hylim/prj**1**.pdf