Scalability Design Patterns

Kanwardeep Singh Ahluwalia
81-A Punjabi Bagh
Patiala - 147001
India
+91 9811016337
kanwardeep@gmail.com

ABSTRACT

This paper presents a pattern language that can be used to make a system highly scalable. This pattern language applies to software systems which need to scale. The pattern language addresses this problem by introducing patterns those touch upon introduction of parallelism to even optimization of algorithms and hardware.

Categories and Subject Descriptors

C.4 [**Performance of Systems**]: Performance attributes, Reliability, availability, and serviceability.

General Terms

Algorithms, Management, Performance, Design, Reliability.

Keywords

Scalability, Parallelism, Algorithm, Multi-thread, Multi-process, Automate, Decentralization, Performance.

1. INTRODUCTION

Scalability is a desirable property of a system, a network, or a process, that 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, e.g. hardware are added.

There are various ways to make a system scalable – some of these ways are driven by the domain to which the system belongs. For example, scientific computing systems may stress faster algorithms. Similarly, a system involving complex calculation may stress faster hardware. On the other hand, 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 and the approach to achieve scalability has been changing. More affordable and abundant hardware has made it possible to focus on adding hardware as one of the simplest ways to enhance scalability, while two decades ago, the focus used to be on faster algorithms, to save the cost of highly priced hardware.

There have also been different perceptions on measuring scalability. In a transaction-oriented system, some would dictate scalability requirements in terms of the maximum number of simultaneous users supported by the system. Others would prefer to dictate the maximum transactions processed per unit of time. For example, a 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 illustrates a way of measuring the scalability of a system; a graph is plotted with load (in terms of simultaneous users) 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 point 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 “cliff point”. This may happen for many reasons, e.g. the system may feel the scarcity of hardware resources, or the system may be experiencing a lot of locking issues which increases the overhead 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.

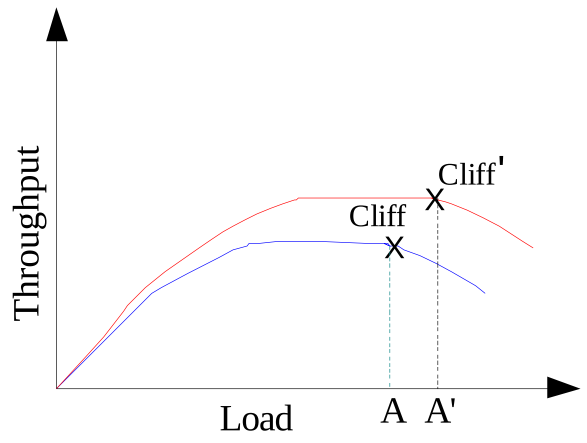

Figure 1: Measuring Scalability

Figure 2 shows system behavior under increasing load after it has been made more scalable, for instance, by adding more processors to the existing hardware. The following figure shows that the system is now able to handle more input load. The 'cliff point' A' has drifted further.

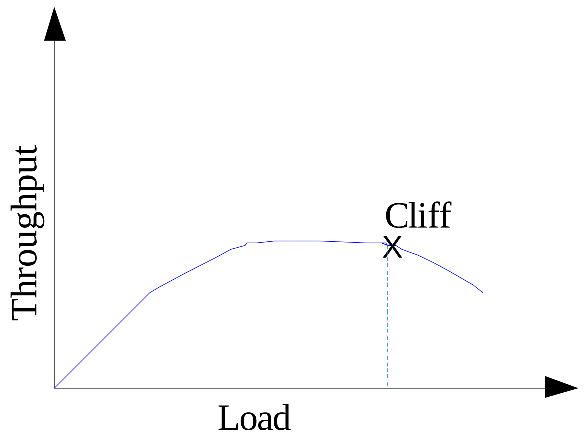

Figure 2: Comparing Scalability

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 we can improve 70% of a module by parallelizing it, and run it on four CPUs instead of one. If • is the fraction of a calculation that is sequential and 1 • • 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 computing 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 to consider while designing a scalable system. These patterns are best suited for transaction-oriented systems. These patterns do not discuss programming techniques that can be used to implement these patterns. The intended audience includes system architects and designers who are designing scalable systems.

2. LANGUAGE MAP

Figure 3 illustrates the relationship between patterns described in this paper. There are two kinds of relationships shown in the language map. The first one is a refinement relationship. The patterns are refined as seen from left to right.

The second kind of relationship shown is a dependency relationship. This kind of relationship exists between patterns which are dependent on each other.

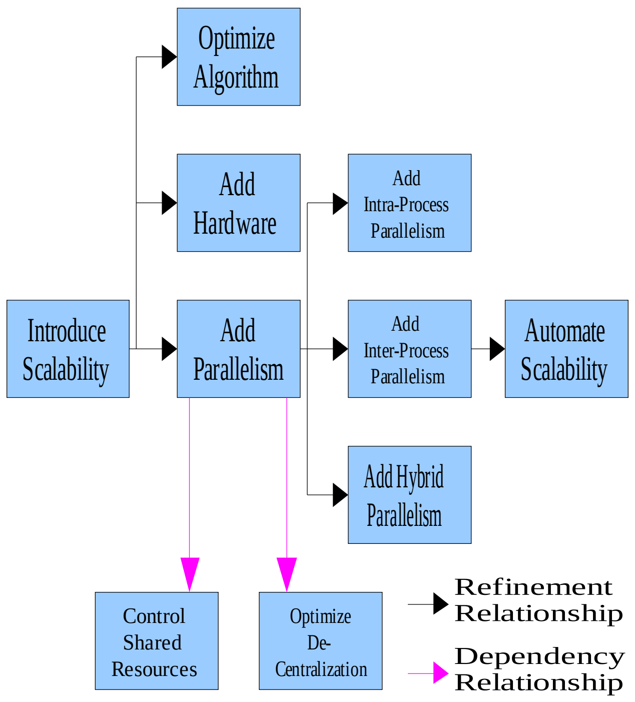

Figure 3: Scalability Patterns Language Map

3. PATTERNS DESCRIPTION

3.1 Pattern 1: Introduce Scalability

3.1.1 Context

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

3.1.2 Problem

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

3.1.3 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.

3.1.4 Solution

The system has to be scalable in order to handle increased load. Many key considerations ranging from application of an optimized algorithm to having optimum amount of decentralization avoiding unnecessary bottlenecks; are required to be made for increased scalability.

Addition of hardware can also result in more resources for processing in the system, which can scale the system to handle the increased load.

The system can also make use of parallelism in order to process the increased load. The parallelism can be introduced with in the process using threads, or system can make use of multiple processes, or a mix of both multi-threaded and multi-processed environment can be used for parallel processing in the system. Further, the system can be made more intelligent to adjust the number of parallel processing units i.e. threads or processes, automatically as per the varying input load.

3.1.5 Resulting Context

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

3.1.6 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.

3.1.7 Related Patterns

Optimize Algorithm
Add Hardware
Introduce Parallelism
Optimize Decentralization
Control Shared Resources

3.2 Pattern 2: Optimize Algorithm

3.2.1 Context

You have decided to **Introduce Scalability** and the system can not **Add Hardware** or **Add Parallelism**.

3.2.2 Problem

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

3.2.3 Forces

- System should be able to 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.

3.2.4 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 that 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 tasks can in some cases be identified with a code walkthrough to mark areas where smarter algorithms can help improving the performance.

If code walkthrough does not help, then certain profiling 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) that are eating up most of the CPU time.

After identifying the problem areas in the code, an alternate algorithm can be used which shall result in keeping the end result same but complete the task in a shorter duration.

3.2.5 Resulting Context

The system is able to maintain its throughput with the increased load, as well as optimized usage of system resources is ensured. However, it may not be always possible or easy to replace the existing algorithm with more efficient one. In that case, the user has to look for other options like **Adding Hardware** or **Parallelism** in the system.

3.2.6 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.

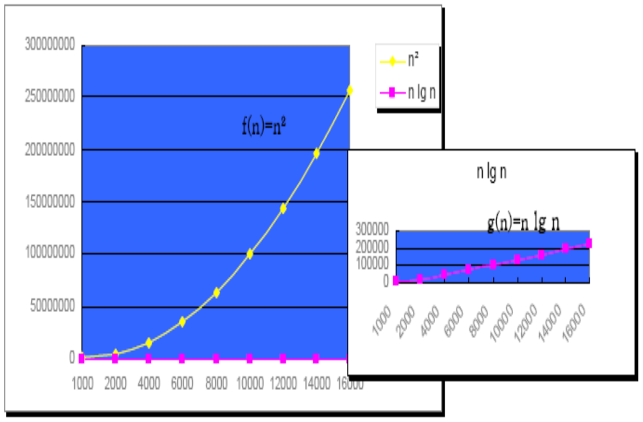

Figure 4 : Comparison of the function of n^2 and n lg n

Another very well known use of enhancing algorithms is by using 'join' in the SQL statements rather than having 'inner' query. 'Inner' query is less scalable as compared to the 'join' query, because 'inner' query results in loading of all the data satisfied by the 'inner' query and then other conditions are looked up on which are specified in the remaining part of the query. However, in case of statements using 'join', all conditions can be specified in one 'where' clause and there is no need to look up for unnecessary data; reducing processing time.

3.2.7 Related Patterns

Introduce Scalability

3.3 Pattern 3: Add Hardware

3.3.1 Context

You have decided to **Introduce Scalability** and the system can not change the code.

3.3.2 Problem

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

3.3.3 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.

3.3.4 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 e.g. prstat/top.

Once the scarce hardware resources are identified, the next step is to add them in efficient quantity. These scarce resources can be added to the existing physical node or altogether a new physical node can be introduced in the system.

Adding hardware to the existing physical creates "vertical scalability". Adding hardware as a separate new node creates enhanced "horizontal scalability".

3.3.5 Resulting Context

The system is able to maintain its throughput with the increased load by having additional hardware. Additional hardware may not always result in the increased scalability for the systems which have inherent bottlenecks due to bad algorithms. Such systems should first **Optimize Algorithm** before seeking additional hardware resources to enhance scalability.

3.3.6 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 if all the existing machines in the cluster are being utilized to their full capacity.

3.3.7 Related Patterns

Introduce Scalability

3.4 Pattern 4: Introduce Parallelism

3.4.1 Context

You have decided to **Introduce Scalability** and the system has the capability to split the work in to pieces that can be executed simultaneously.

3.4.2 Problem

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

3.4.3 Forces

- System should be able to process multiple tasks at the same time.
- System should be able to maintain its transaction processing rate with increasing input load.
- 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.

3.4.4 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.

Divide the work in to tasks that can be done simultaneously to do more processing in the same time. The longer the task, the more it improves the scalability. Parallelism will also ensure optimized usage of system resources such as a CPU.

Further parallelism can be introduced in a way to process multiple transactions simultaneously as shown in the **Figure 5** 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. In the later case, each parallel processing unit would be a specialized one and may not be similar to the others.

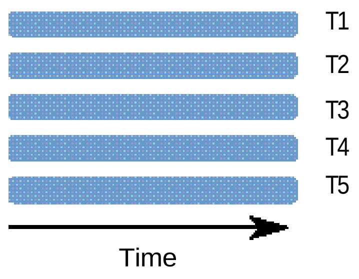

Figure 5 : Multiple Simultaneous Transactions

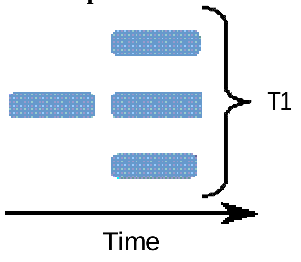

Figure 6 : Single transaction split in to multiple parallel tasks

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

3.4.5 Resulting Context

The system is able to maintain its throughput with the increased load, and ensure optimized usage of system resources is ensured. The down side of adding parallelism is that it usually makes a system complex to maintain and debug. A bad design will make it more prone to defects and can result in losing the integrity of data. Hence, it is usually not the first choice of many designers to enhance the scalability of the system.

3.4.6 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**.

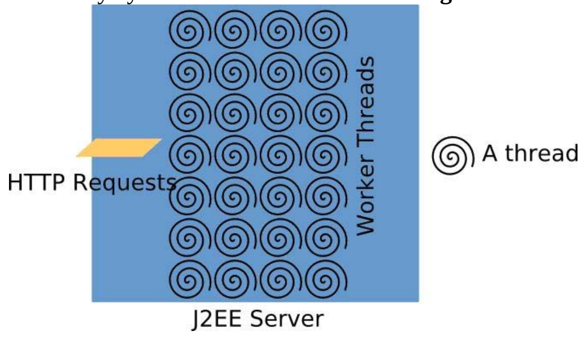

Figure 7 : Worker threads in an application server

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

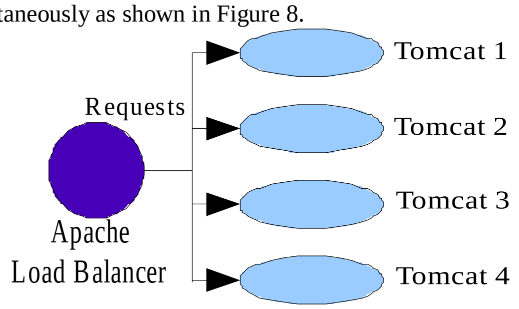

Figure 8 : Apache - Tomcat Cluster

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

3.4.7 Related Patterns

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

3.5 Pattern 5: Optimize De-centralization

3.5.1 Context

You have decided to **Introduce Parallelism** and the system is not able to scale due to bottlenecks (the areas related to resources those are required by parallel processing paths).

3.5.2 Problem

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

3.5.3 Forces

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

3.5.4 Solution

Design the system to avoid bottlenecks. Bottlenecks should be avoided by following a 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.

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. The most common technique for removing the scaling bottlenecks involves providing an individual copy of the resource that is creating the bottleneck to each contender. Other approaches include minimizing the bottleneck area, so that the impact of bottleneck is reduced.

It is not possible to completely get rid of centralized processing. Instead, it may be beneficial in some places, predicting bottlenecks are hard, but measuring them is easy. Therefore, take an incremental approach to optimize decentralization, wherein bottlenecks detected during processing should be removed and then the system is again observed for further bottlenecks. This ensures that only centralized processing areas posing as bottlenecks are removed.

3.5.5 Resulting Context

The system is able to do parallel processing without running in to scaling bottlenecks. The user needs to be careful while using this pattern by not removing useful centralizations, which may break the system logic. Centralization is at times necessary e.g. even though there are various state governments running the states, the role of Federal/Central government is important to bring synergies between these governments.

3.5.6 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.

3.5.7 Related Patterns

Introduce Parallelism

3.6 Pattern 6: Control Shared Resources

3.6.1 Context

You have decided to **Introduce Parallelism** and the system has to access some of the shared resources while doing parallel processing.

3.6.2 Problem

How system should be able to have parallel processing without corrupting the shared resources?

3.6.3 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.

3.6.4 Solution

Identify the shared resources in the system and categorize them as “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 way to prevent the corruption of shared resources is to acquire a lock on the shared resource, modify it, and release the lock. This ensures that the shared resource is being modified by only one parallel processing unit at a time.

Using locks is not as trivial as it sounds. Locks should be carefully used to avoid deadlocks that may occur due to following conditions.

- A parallel processing unit already holds the lock and requests the same lock again.
- Two or more parallel processing units form a circular chain where each waits for a lock held by the next in the chain.

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

3.6.5 *Resulting Context*

The system is able to access its shared resources without corrupting them while doing parallel processing. However, as discussed above the shared resources have to be accessed carefully so as to avoid their corruption and deadlock conditions.

3.6.6 *Known Uses*

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

3.6.7 *Related Patterns*

Introduce Scalability

3.7 Pattern 7: Add Intra-process parallelism

3.7.1 *Context*

You have decided to **Introduce Parallelism** and the system has only one process available to handle the increasing input load.

3.7.2 *Problem*

How can a system be able to execute tasks in parallel when there is only one process in the system?

3.7.3 *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.

3.7.4 *Solution*

Use 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 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 in input load.

Multi-threading will also allow the system overlap various tasks independent of each other, e.g. one thread can communicate with the disk while the other thread can communicate with the network. Multi-threading also allows overlapping a CPU-intensive task with a non-CPU intensive task. One thread could be receiving data from the network while the other could be parsing 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; whereas the process with Intra-process parallelism is shown on the right hand side of the dotted line with multiple threads accessing more than one CPU at a time.

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

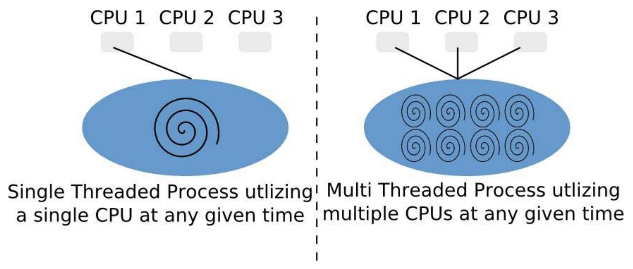
CPU 1 CPU 2 CPU 3
Single Threaded Process utlizing a single CPU at any given time
CPU 1 CPU 2 CPU 3
Multi Threaded Process utlizing multiple CPUs at any given time

Figure 9 : Display of Intra-Process Parallelism

3.7.5 *Resulting Context*

The system is able to do parallel processing even from with in a single process by making use of threads. Threads add complexity to the system. It is difficult to maintain and debug a multi-threaded system. Hence, they should be used with care and only if required. Also, adding a 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.

3.7.6 *Known Uses*

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

3.7.7 *Related Patterns*

Automate Scalability

3.8 Pattern 8: Add Inter-process parallelism

3.8.1 *Context*

You have decided to **Introduce Parallelism** and the system can not scale using **Intra-process parallelism**.

3.8.2 *Problem*

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

3.8.3 *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.

3.8.4 *Solution*

Replicate processes by spawning multiple instances. All 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 that helps in assigning the task to each process.

The structure is shown below in the **Figure 10**. On the left hand side of the dotted line is a single process for handling entire load sent from the client. On the right hand side is a system with multiple processes to handle the load. A load balancer loads the multiple processes in the system. The client is not aware of the multiple processes in the system, but definitely enjoys the enhanced scalability.

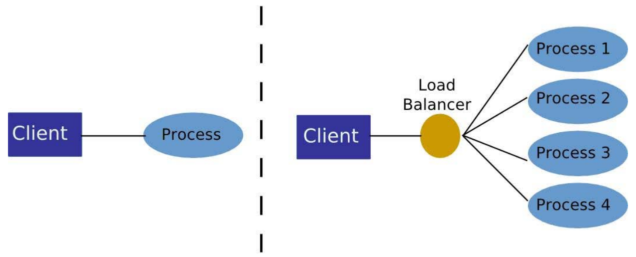

Figure 10 : Effect of introducing Inter Process Parallelism

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

3.8.5 *Resulting Context*

The system is now able to handle the increased load by spawning its multiple processes without compromising on the throughput. However, having too many processes may not make the system more scalable, because the overhead introduced due to coordination between the processes may takeover the gain from having multiple processes. Hence, the system should only introduce an 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.

3.8.6 *Known Uses*

The Apache-Tomcat cluster is an 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 an asynchronous message queue like JMS. Here the main processes delegate the logging to a centralized logger without waiting for actual logging to happen, these processes move on to process the remaining transaction.

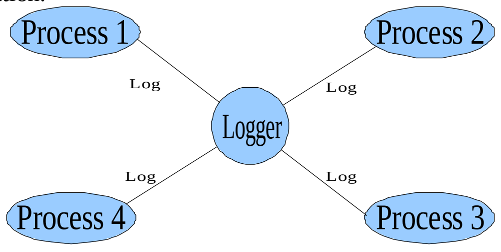

Figure 11 : Centralized Logger

3.8.7 *Related Patterns*

Automate Scalability

3.9 Pattern 9: Add Hybrid parallelism

3.9.1 *Context*

You have decided to **Introduce Parallelism** and the system is capable of being both multi-threaded and multi-processed.

3.9.2 *Problem*

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

3.9.3 *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.

3.9.4 *Solution*

Spawn multiple process instances as specified in the pattern “Inter-process Parallelism” after a process is not able to scale by increasing the number of threads as specified in “Intra-process Parallelism”. This is depicted in **Figure 12**.

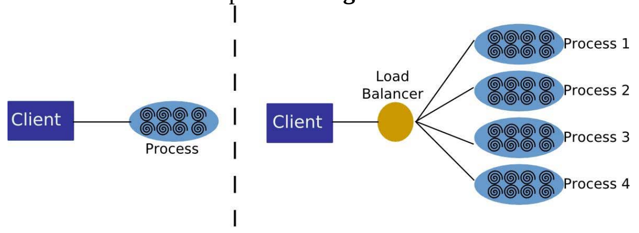

Figure 12 : Hybrid Parallelism

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.

3.9.5 *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.

3.9.6 *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.

3.9.7 *Related Patterns*

Automate Scalability

3.10 Pattern 10: Automate Scalability

3.10.1 *Context*

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

3.10.2 *Problem*

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

3.10.3 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.

3.10.4 Solution

Use a monitoring entity that measures the current throughput with the ability to increase or decrease the number of threads or processes in the system.

When 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 the Intra or Hybrid parallel system) or the number of processes (in the Inter or Hybrid parallel system). This should increase the transaction rate. The number of threads or processes should not be increased once the system increases its throughput to the desired rate.

Additionally, the monitoring entity should decrease the number of processes or threads when 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 a Last in First out (LIFO) manner.

3.10.5 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.

3.10.6 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.

3.10.7 Related Patterns

Intra Process Parallelism
Inter Process Parallelism
Hybrid Parallelism

4. ACKNOWLEDGMENTS

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.

Credit also goes to the participants of writer’s workshop at PLoP’07 who gave very useful comments.

5. REFERENCES

[1] D.H. Brown Associates, Inc. 2004. Vertical and Horizontal Scalability. http://www.sun.com/servers/wp/docs/dhbrown.scalability.final.4.2004.pdf

[2] Hye Eun, Lim. Sort Comparisons at http://www-users.cs.umn.edu/~hylim/prj1.pdf

[3] The Apache Software Foundation. Clustering/Session Replication HOW-TO. http://tomcat.apache.org/tomcat-5.5-doc/cluster-howto.html

[4] Xian-He Sun Rover, D.T. Scalability of parallel algorithm-machine combinations. http://ieeexplore.ieee.org/xpl/freeabs_all.jsp?arnumber=285606&fromcon