Building blocks of a Scalable Architecture

Reproduced from https://dzone.com/articles/component-load-testing

Introduction

Modern online application development requirements are driven by the need for a highly-scalable and performance-centric platform. Like System Functionality Requirements, the NFR's (Non Functional Requiments) like scalability, performance and availability are given equal importance. For many years IT industry has been struggling to build highly scalable systems. In this struggle it has learned many good architecture and design principles.

This document captures some of these learning which are most frequently used from a very high level view. These learning has been categorized into design principles, design rules, design patterns, design antipatterns and building blocks of highly scalable online platforms. Followings are the high level definitions of these.

- **Design Principles** are the fundamental design laws to be followed to build scalable systems. Design rules, patterns and anti-patterns are derived from one or more of these principles.
- **Design Rules** are second level design laws that tells you of what to do and what not to do based on the past learning or what worked and what did not work.
- **Design Patterns** are general reusable solutions that have been discovered in the past for building scalable systems.
- **Design Anti Patterns**: are common design solutions which are proved to be ineffective for building scalable systems.
- **Building blocks**: are commonly used infrastructure software, tools, frameworks, and services that can be used to build a scalable system.

Principles

Scalability principles are basic proposition, behavior, and properties of scalable systems. All scalability design patterns, rules, and anti-patterns are derived from these principles. If understood and used rationally we can design scalable systems without learning lot of intricacies and details of scalable systems. Before choosing any architectural and design option, consider these principles.

Simplicity

of design not only simplifies the scalability but also simplifies development, deployment, maintenance and support.

Decomposition

Decompose the system into smaller manageable subsystems. Each subsystem can carry out independent function. The subsystems should be able to independently run in a separate process or threads and enabled to scale using various load balancing and other form or tuning techniques.

Asynchronous

Asynchronous processing enables process execution without blocking on resources. Asynchronous prcessing comes with overhead as it is relatively complex to design and test.

Loose Coupling High Cohesion

Reducing coupling and increasing cohesion are two key principles to increase application scalability. Coupling is a degree of dependency at design or run time that exists between subsystems. Coupling can limit scalability due to server and resource affinity. In addition, loose coupling provides greater flexibility to independently choose optimized strategies for performance and scalability for different subsystems.

Weak cohesion among subsystems tends to result in more round trips because the classes or components are not logically grouped and may reside in different tiers. This can force you to require a mix of local and remote calls to complete a logical operation. So the sequence of interactions between subsystems becomes complex and chatty which reduces the scalability. therefore, each subsystem should be designed to work independently with minimum dependencies with other subsystems.

Concurrency and Parallelization

Concurrency is when multiple tasks performed simultaneously with shared resources. Parallelization is when single task divided into multiple simple independent tasks which can be performed simultaneously.

Parsimony

Parsimony means that an architect and developer must be economical towards the system resources in their design and implementations. They should try to use system resources (CPU, disk, memory, network, database connection etc) as effectively and efficiently as possible. It also means that scarce resources must be used carefully. Such resources might be cached or pooled and multiplexed. This principle pervades all the other things. No matter how well a system is architeched and designed, if system resources are not used carefully, application scalability and performance suffers.

Decentralization/Distribution

By definition a distributed system is a collection of subsystems running on independent servers that appears to its users as a single coherent system. Distributed systems offers high scalability and high availability by adding more servers.

Rules

Here are some common design rules derived from design principles:

- Ensure your design works if scale changes by 10 times or 20 times;
- Do not use bleeding edge technologies;
- Optimize the design for the most frequent or important tasks;
- Design for horizontal scalability;
- Design to use commodity systems;
- Design to Leverage the Cloud;
- Use caches wherever possible;
- Design scale into your solution;
- Simplify the solution;
- Performing I/O, whether disk or network, is typically the most expensive operation in a system;
- Transactions use costly resources;
- Use back of the envelope calculations to choose best design.

Antipatterns

Description	Problem	Solution
Excessive processing	Consumes resources which can be used by other transactions. Increases response time Decreases throughput	Remove, postpone (asynchronous), prioritize, or reorder the processing step Leverage caching to reuse loaded or calculated data
Presentation of large set of data to users	Typically OLTP users do not consume large amount of data so it is wastage of processing resources. Severe system performance issue can arise depending on the volume of data	Paginate results Narrow down search criteria
Not releasing resource immediately after use	Cause of connection leaks, deadlocks, performance degradation and other unexpected behavior Threads, sockets, database connection, file handler, and other resources can be a victim	Use finally block to ensure release Release resource as quickly as possible
Inefficient UI design	Additional work for the system Unnecessary large data presentation to user	Refractor UI

	More number of pages presentation	
Unnecessary data retrieval	Unnecessary database, disk and network resources utilization	Select only fields and rows from database which are required.
Incompatibility with HA deployment	Application should be designed to support high availability and multi instance deployment.	
Too many objects in memory	Unnecessary memory utilization Long garbage collection pause	Use singleton design pattern Create stateless services Reuse data objects Set cache expire time
No data caching	Unnecessary remote calls Unnecessary processing to calculate or transform data Increase database load Slow performance and scalability	Cache most frequently and read mostly data. Cache complex objects graphs to avoid processing.
Unlimited Resource usages		

Usage of outdated tools/technology/API		
Poor database schema design.	Expensive SQL that do not scale.	Do not over normalize database. Create summary tables for reports Use NoSQL database if required.
Poor transaction design.	Can cause locking and serialization problems.	Avoid transactions as much as possible. Avoid distributed transaction Avoid long lived transactions
Scaling through 3rd parties software	If you are relying on a vendor for your ability to scale such as with a database cluster you are asking for problems.	Use clustering and other vendor features for availability, plan on scaling by dividing your users onto separate devices, sharding.
Monolithic databases	If your data get big enough you will need the ability to split your database.	
Underestimated network latency		

Excessive layering	Each layer creates many temporary objects e.g. DTOs, consumes processing for data transformation and consumes network bandwidth if layers are spread across servers.	Simplify design
Excessive network round tripping, **Chatty Services**	High network usage Slow response	Design coarse grain services Design stateless services Try to query data from database with minimum number of interactions. Avoid fetching unnecessary data from database Cache data or service responses wherever possible
Overstuffed Session	Will use large memory even for inactive users till the session is destroyed. Application server will be able to handle less number of concurrent users. If we are using application server clustering there would be lot of network overhead.	Design application as stateless as possible. Use cache, Cookies, hidden fields, URL query parameters etc.

Design Patterns

Shared nothing architecture	Shared nothing architecture (SNA) is horizontal scalability architecture. Each node in SNA has its own memory, disks and input/output devices. Each node is self sufficient and shares nothing across the network. This reduces the any kind of contention among nodes as there is no scope for data or any other kind of resource sharing. This type of architecture is highly scalable for web applications. SNA partition its different layers (Web server, App Server, DB) to handle the incoming user requests based on many different policies such as geographic area, type of users etc. Google has implemented this which has enabled it to scale its web applications effectively by simply adding nodes.
Database sharding	Database sharding is a shared nothing horizontal database partitioning design pattern. Each database shard can be placed on separate machine or multiple shards can reside on single machine. This distributes data on multiple machines which means that database load is spread out on multiple machines which greatly improves the performance and scalability. Some of the advantages of sharding are Massive scalability, High availability, Faster queries; More write bandwidth, reduced cost as databases can run on commodity servers.
Master slave database	If your application is read heavy and does not require horizontal write scalability you can use master slave database replication. Many popular database provides this feature out of the box e.g. MySQL, Postgres etc.
Optimistic concurrency **Optimistic locking**	

Eventual Consistency	
Pooling and multiplexing	Pooling is an effective way to use expensive resources for example, large object graphs, database connections, threads.
Near real-time synchronizatio n of data	A delay of few seconds and more should be acceptable for most of the integration systems so convert real time synchronous distributed transactions into near real time asynchronous one.
Just-enough data distribution	Distribute out as little data as possible. For an object to be distributed outward, it must be serialized and passed through memory or over a network. This involves three system resources: CPU utilization and memory in the server to serialize the object and possibly packetize it for travel across the network, network bandwidth or interprocess communication activity to actually transmit to the receiver, CPU utilization and memory in the receiver to (possibly) unpacketize, deserialize, and reconstruct the object graph. Hence, an object's movement from server to receiver comes at a fairly high cost.
Use compression before sending data over a network	
Data archival	Keep current most frequently used online data separate from old less frequently data. For this you may need to refractor UI.
Intelligent load distribution	Incoming HTTP requests redirect to the mirrored facilities based on some combination of available server and network capacity. This can be accomplished internally or by subscribing to one of the commercial providers who specialize in this type of service.

Graceful service degradation	
Map reduce	
Autonomy	The system is designed such that individual components can make decisions based on local information.
Failure tolerant:	The system considers the failure of components to be a normal mode of operation, and continues operation with no or minimal interruption.
Collocation	Reduce any overheads associated with fetching data required for a piece of work, by collocating the data and the code.
Caching	If the data and the code can't be collocated, cache the data to reduce the overhead of fetching it over and over again.
Remoting	Reduce the amount of time spent accessing remote services by, for example, making the interfaces more coarse-grained. It's also worth remembering that remote vs local is an explicit design decision not a switch and to consider the first law of distributed computing - do not distribute your objects.
load balancing	Spreading the load across many instances of system/subsystem/component for handling the requests.
Queuing	

Batch processing	Achieve efficiencies of scale by processing batches of data, usually because the overhead of an operation is amortized across multiple request
Relax data constraint	Many different techniques and trade-offs with regards to the immediacy of processing / storing / access to data fall in this strategy

Building Blocks of Scalability

Load Balancers	Hardware load balancers Software load balancers
Messaging queues	MSMQ, MQSeries
Caches	Memcache, ehcache
Reverse proxies	Squid, Varnish
CDN	Akamai
NoSQL databases	MOngo
Embeded databases	SQL Express
Cloud	AWS, Azure
Language features	Concurrency, queues, locks, asynchronous, thread pools
Frameworks	Akka, Storm, Kafka