Software System Architecture and Design

The Performance and Scalability Perspective
The Performance and Scalability Perspective

- **Desired Quality**: The ability of the system to:
  - predictably execute within its mandated *performance* role and
  - to *handle increased processing volumes* in the future if required.

- **Applicability**:
  - Any system with complex, unclear, or ambitious performance requirements;
  - systems where future expansion is likely to be significant.
## Applicability to Views

<table>
<thead>
<tr>
<th>View</th>
<th>Applicability</th>
</tr>
</thead>
<tbody>
<tr>
<td>Functional</td>
<td>Performance requirements may require to make changes in the Functional View (e.g., to consolidate system elements to avoid communication overhead).</td>
</tr>
<tr>
<td>Information</td>
<td>Considering scalability may suggest elements of the Information view that could be replicated or distributed in support of this goal.</td>
</tr>
<tr>
<td>Concurrency</td>
<td>Considering performance and scalability may result in concurrency becoming a more important design element.</td>
</tr>
<tr>
<td>Development</td>
<td>A set of guidelines related to performance and scalability that should be followed during software development.</td>
</tr>
<tr>
<td>Deployment</td>
<td>Clusters, grids, high-performance hardware, etc.</td>
</tr>
<tr>
<td>Operational</td>
<td>The application of this perspective highlights the need for performance monitoring and management capabilities.</td>
</tr>
</tbody>
</table>
Concerns: Response Time

- Response time is the length of time it takes for a specified interaction with the system to complete.
- Two classes of response times:
  - Responsiveness considers how quickly the system responds to routine workloads such as interactive user requests (the order of a few seconds).
    - The key consideration for such workloads is user productivity, ensuring that the system does not slow down its users.
  - Turnaround time is the time taken to complete (turn around) larger tasks (few minutes or hours).
    - The key considerations are whether the task can be completed in the time available to it and the impact the task has on the system responsiveness while it is running.
- One of the things that make performance complicated is the number of event sources and arrival patterns.
Concerns: Response Time:

Event arrival patterns

- An arrival pattern for events may be characterized as either periodic, stochastic or sporadic.
- For example, a periodic event may arrive every 10 milliseconds.
  - Periodic event arrival is most often seen in real-time systems.
- Stochastic arrival means that events arrive according to some probabilistic distribution.
  - The number of incoming declarations to Tax Inspectorate (VMI)
- Events can also arrive sporadically, that is, according to a pattern not capturable by either periodic or stochastic characterizations.
  - The number of users performing searches in Google
Concerns: Response Time: Multiple users

- Multiple users or other loading factors can be modeled by varying the arrival pattern for events.
  - In other words, from the point of view of system performance, it does not matter whether:
    - one user submits 20 requests in a period of time or
    - whether two users each submit 10.
  - What matters is the arrival pattern at the server and dependencies within the requests.
Concerns: Response Time Characteristics

- The response of the system to a stimulus can be characterized by:
  - *latency* (the time between the arrival of the stimulus and the system's response to it),
  - *deadlines in processing* (in the engine controller, for example, the fuel should ignite when the cylinder is in a particular position, thus introducing a processing deadline),
  - the *throughput* of the system (e.g., the number of transactions the system can process in a second),
  - the *jitter* of the response (the variation in latency),
  - the number of events not processed because the system was too busy to respond, and
  - the data that was lost because the system was too busy
Concerns: Throughput

- Throughput is defined as the amount of workload the system is capable of handling in a unit time period.
  - Throughput and response time have a complex interrelationship in most systems.
- In general, the shorter your transaction processing time, the higher the throughput your system can achieve.
  - However, as the load on the system increases (and throughput rises), the response time for individual transactions tends to increase.
- Therefore, it is quite possible to end up with a situation where throughput goals can be met only at the expense of response time goals, or vice versa.
Concerns: Scalability

- Scalability is the ability of a system to handle:
  - the increased workload, which may be due to an increase in the number of requests, transactions, messages, or jobs the system is required to process per unit of time or
  - an increase in the complexity of these tasks.

- Methods of adding more resources for a particular application fall into two broad categories:
  - To *scale horizontally* (or *scale out*) means to add more nodes to a system, such as adding a new computer to a distributed software application.
    - Many computers may be configured in a *cluster*.
  - To *scale vertically* (or *scale up*) means to add resources to a single node in a system, typically involving the addition of CPUs or memory to a single computer.
Scaling Out:

Computer Clusters

- A computer cluster consists of a set of loosely connected computers that work together so that in many respects they can be viewed as a single system.
  - The components of a cluster (nodes) are usually connected to each other through fast local area networks, each node running its own instance on an operating system.
- Clusters are usually deployed to improve performance and availability over that of a single computer,
  - while typically being much more cost-effective than single computers of comparable speed or availability.
- "Load-balancing" clusters are configurations in which cluster-nodes share computational workload to provide better overall performance
- "High-availability clusters" (also known as failover clusters, or HA clusters) improve the availability of the cluster approach.
  - They operate by having redundant nodes, which are then used to provide service when system components fail.
  - HA cluster implementations attempt to use redundancy of cluster components to eliminate single points of failure.
  - We will talk about HA clusters in the Availability Perspective.

The Performance and Scalability Perspective
Scaling Out: Clusters: **Load balancing**

- *Load balancing* is a computer networking methodology to distribute workload across:
  - multiple computers,
  - network links,
  - central processing units,
  - disk drives, or
  - other resources,

  to achieve:
  - optimal *resource utilization*,
  - maximize *throughput*,
  - minimize *response time*, and
  - avoid *overload*.
Scaling Out:

Load balancer features

- **Sticky sessions** are used to direct request to correct node if system uses sessions to store transient user data (e.g. shopping cart).
- **Asymmetric load**: A ratio can be manually assigned to cause some backend servers to get a greater share of the workload than others.
  - This is sometimes used as a crude way to account for some servers having more capacity than others and may not always work as desired.
- **Priority activation**: When the number of available servers drops below a certain number, or load gets too high, standby servers can be brought online.
- **SSL Offload and Acceleration**
- **Distributed Denial of Service (DDoS) attack protection**: load balancers can provide features such as **SYN cookies** and delayed-binding (the back-end servers don't see the client until it finishes its TCP handshake) to mitigate **SYN flood** attacks.
- **Health checking**: the balancer will poll servers for application layer health and remove failed servers from the pool.
- many others.
Concerns: Predictability

- Predictability means that similar transactions complete in very similar amounts of time regardless of when they are executed.
  - Similarly, the maximum transaction throughput the system can cope with should not vary significantly over time (in particular, it shouldn’t decrease).
- Predictability is often a more desirable quality than absolute performance.
Concerns: Peak Load Behavior

- Nearly all computer systems eventually exhibit poor performance as the load on them increases.
- This behavior is usually caused by one or more critical resources in the system becoming so overloaded that it can no longer work effectively.
  - for example, a network card is so swamped by incoming connection requests that it cannot service any of them effectively.
- *Stress-testing* helps to reveal the “knee” point.
Architectural Tactics (1 of 4)

Let's start with simple ones...

- **Reduce Contention via Replication**
  - A possible solution for some contention problems is to replicate system elements—hardware, software, or data—a tactic that you will often need to combine with the related tactics of partitioning and parallelizing.

- **Consolidate Related Workload**
  - Consolidate related tasks into batches and process groups of related requests together.
  - This pattern of processing normally allows a single initialization step, a number of operation processing steps, and then a single tear-down step—thus saving the initialization and teardown steps that would be required for each operation if processed separately.

- **Minimize the Use of Shared Resources**
  - Use techniques such as *hardware multiplexing* to eliminate hardware hot spots in your architecture.
  - Favor *short, simple transactions* over long, complex ones where possible (because transactions tend to lock up resources for extended periods).
  - *Do not lock resources in human time* (e.g., while waiting for a user to press a key).
Simple: Degrade Gracefully

- There could be proactive monitoring in place that ensures that failures are recognized quickly and handled reliably—in particular, overloading and failures should not ripple destructively throughout the system.
- One way of achieving this is to design the system so that it contains the software equivalent of “circuit breakers” that prevent internal components from becoming overloaded.
- Other common tactics are:
  - to reject workload when the system is overloaded, perhaps asking the user to try again in a few minutes, and
  - to throttle demand by introducing timeouts for service calls made to the system.
Architectural Tactics (3 of 4)

Hard: Partition and Parallelize, Use Asynchronous Processing

- If your system involves lengthy process, to reduce the response time:
  - partition the process into a number of smaller processes,
  - execute these subprocesses in parallel,
  - consolidate their output into a single result.

- Patterns / technologies / paradigms:
  - Concurrency patterns
    - Producer-Consumer, Work pool, Map-Reduce, Fork-Join, etc.
  - EJB/Spring Framework: @Asynchronous
  - Java SE: CompletableFuture
    - Composing several CompletableFutures allows the construction of pipeline-like recipes for multiple tasks, with control over asynchronous execution
    - Functional-Style Callbacks Using Java 8's CompletableFuture
  - Actor Model paradigm
    - Akka (http://akka.io/)
    - Orleans (http://dotnet.github.io/orleans/)
    - Erlang (http://www.erlang.org/)
Architectural Tactics (4 of 4)

Hard:

- **Scale Up or Scale Out**
  - The problem with achieving scalability by adding hardware (particularly when scaling out) is that it usually isn’t very effective unless the system has been designed to take advantage of the new hardware.

- **Relax Consistency Constraints**
  - As long as an individual consumer sees consistent data, the need for immediate system-wide consistency is something that can be relaxed, as long as it happens “eventually”
    - [Eventual consistency](#) (more details in the following slides)
  - Split single database to two: Operational and Reporting: Reporting Database Pattern
    - Reports become eventually consistent
  - Separate read and write data models: CQRS Pattern
    - All data become eventually consistent
Liability:

- Reports become eventually consistent!
CQRS Pattern
(Command-Query Responsibility Segregation)
CQRS Pattern

  - The query model for reading data and the update model for writing data **may access the same physical store**
  - perhaps by using SQL views
  - However, it is common to **separate** the data into different physical stores to maximize performance and scalability
Fundamentals of Scalability

- Next we will discuss:
  - Why some applications do not scale
    - Amdahl's law
  - What is linear scalability
  - How to achieve linear scalability
Scalability limits: **Amdahl's law**


- Predicts maximum expected improvement to an overall system when only part of the system is improved.
  - It is often used in parallel computing to predict the theoretical maximum speedup using multiple processors.
- If:
  - $P$ is the proportion of a program that can be made parallel (i.e., benefit from parallelization), and
  - $(1 - P)$ is the proportion that cannot be parallelized (remains serial),
- then the maximum speedup that can be achieved by using $N$ processors is:

$$S(N) = \frac{1}{(1 - P) + \frac{P}{N}}$$
P versus 1-P
Amdahl’s Law

- Parallel Portion
  - 50%
  - 75%
  - 90%
  - 95%

Speedup vs. Number of Processors
Concerns: Scalability

(The Scalability Revolution: From Dead End to Open Road)

- Having *scalability* means that your application can increase its working capacity while keeping three things constant:
  - *Consistency* of the business logic and data—otherwise when you scale, things won’t work. (If scaling compromises reliability, but to an extent that your users can tolerate, it can still count as scaling.)
  - *End-to-end latency* – otherwise when you scale, you might be supporting more clients, but degrading the service level each client receives. (If scaling degrades performance, but the service level is still acceptable to your users, it can still count as scaling.)
  - *The application code* – otherwise you aren’t scaling the application, you’re replacing it.
Scalability Barriers
(The Scalability Revolution: From Dead End to Open Road)

- Scalability **crash barriers**:
  - **Consistency barrier**—occurs when you cannot scale the application any more without threatening the level of business logic reliability demanded by your users.
  - **Latency barrier**—occurs when you cannot scale the application any more without increasing end-to-end latency to an unacceptable level.
  - **Coding barrier**—occurs when you cannot scale the application any more without significantly rewriting the code.

- A situation in which scaling is possible, but the next capacity unit is too expensive to justify, can be called a **marginal cost barrier**.
  - Here it is possible to scale in principle – *it’s just not worth it*
Scalability Metrics

(The Scalability Revolution: From Dead End to Open Road)

- If you need to **double** your capacity (concurrent users, incoming messages, etc.) tomorrow, while maintaining current latency and consistency levels, will you need to provide:
  - twice the resources you have today, or
  - ten times the resources?

- The difficulty of scaling can be better understood by defining *scalability metrics*. If capacity is a function \( C(x) \) of hardware, then:
  - **First derivative of scalability** ( \( C'(x) \) ) – how many units of hardware do you need to add to get one additional unit of capacity?
  - **Second derivative of scalability** ( \( C''(x) \) ) – as you scale, does each additional unit of capacity require a bigger increase of hardware resources, and by how much?
Linear Scalability – No Barriers
(The Scalability Revolution: From Dead End to Open Road)

- **Linear scalability** occurs when each new hardware unit always contributes the same amount of additional capacity.
  - Linear scalability is possible only if there are no scalability barriers at any point along the application’s scaling path.

- In case of linear scalability:
  - the first scalability derivative is **constant**,
  - the second scalability derivative is **zero**.
Universal Scalability Law (USL)

http://www.perfdynamics.com/Manifesto/USLscalability.html

\[ C(N) = \frac{N}{1 + \alpha (N - 1) + \beta N (N - 1)} \]

- \( N \) – number of processors (nodes); \( C(N) \) – capacity
- \( 0 \leq \alpha, \beta < 1 \)
- \( \alpha \) – level of contention (e.g., proportion that cannot be parallelized)
- \( \beta \) – coherency delay (i.e., latency for data to become consistent) in the system (amount of point-to-point interactions)
  - Unfortunately, Gunther himself provides little help in telling us how to estimate \( \beta \)
- The \( \beta \) parameter also quantifies the retrograde throughput seen in many stress tests but not accounted for in Amdahl's law
  - If \( \beta = 0 \) we get Amdahl's law
Universal Scalability Law (USL)

http://www.perfdynamics.com/Manifesto/USLscalability.html

A. Equal bang for the buck

B. Cost of sharing resources

\[ \alpha = 0, \beta = 0 \]

\[ \alpha > 0, \beta = 0 \]
Universal Scalability Law (USL)

http://www.perfdynamics.com/Manifesto/USLscalability.html

C. Diminishing returns from contention

D. Negative returns from incoherency

\[ \alpha \gg 0, \beta = 0 \]

\[ \alpha \gg 0, \beta > 0 \]
Tier-Based == Non-Linear Scalability

(The Scalability Revolution: From Dead End to Open Road)

- The system is as latent as its weakest link. **Tier-based system:**

  - If:
    - maintainers choose to apply the scalability fixes, the application grows more and more complex, until it hits a **marginal cost barrier**;
    - but if they don’t apply these fixes, the application very quickly hits a consistency or latency **crash barrier**, and must be replaced.

- According to Amdahl’s law, if as little as **10% of computing power is invested in overhead**, then a hundred-fold increase in processing power yields only **9.17 times** the speed improvement.
Tier-Based == Non-Linear Scalability

- Most existing applications are stateful and, therefore, by definition have a requirement for synchronization as part of their code.
  - This means that the throughput gain expected for these types of applications by throwing more hardware at the problem is going to be fairly low.
- In other words, if you want to achieve true linear scalability in a stateful environment you must design your application in a distributed/partitioned fashion.
Achieving Scalability

- **The Reactive Manifesto**
  - event-driven,
  - scalable,
  - resilient and
  - responsive

- **Other/similar approaches/ideas:**
  - [Microservices](#),
  - [Event-driven architecture](#) (EDA),
  - Staged event-driven architecture,
  - [Actor model](#)
Scalability, Availability and Stability Patterns
Homework

- Book "Software Systems Architecture"
  - Chapter 26 The Performance and Scalability Perspective