UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

COS301 Mini Project Functional Architecture Requirements

Group Name: Group 7_a

Roger Tavares *1*0167324
Thinus Naude *1*3019602
Kabelo Kgwete *1*1247143
Sylvester Mpanganer *1*1241617
Maphuti Setati *1*2310043
Ruth Ojo *1*2042804
Axel Ind *1*2063178
Lindelo Mapumulo *1*2002862
Maria Qumalo *2*9461775

Git repository link:
`https://github.com/thinusn/COS301MiniProjectArchitectureRequirements`

Final Version

March 9, 2015

Contents

1 Introduction

This document was compiled by our group during our meetings and was produced as a whole by the team.

This document contains specifications of the software architecture requirements. This is the infrastructure upon which the application functionality will be developed. The following non-functional requirements are addressed in depth with supporting diagrams (when necessary):

- Access and Integration requirements.
- Architectural responsibilities.
- Quality requirements.
- Architecture constraints as specified by the client.

2 Architecture requirements

2.1 Architectural scope

2.2 Critical quality requirements

2.2.1 Scalability

Description

Scalability is an essential aspect of a system and is the ability of a system to be easily enlarged in order to accommodate a growing amount of work.

Justification

As the Buzz System should allow for more than a million concurrent users, the system must be able to handle that large number without breaking down or reducing performance.

Mechanism

1. Strategy:

   Scalability can be achieved by:

   - Clustering: using more resources by running many instances of the application over a cluster of servers or instances.
   - Efficient use of storage: data storage can be efficiently used through compression of the data (reducing data size to make room for more) paging (ensuring that primary storage is used only for more crucial data) as well as de-fragmentation (organizing the data into continuous fragments and free more storage space). by ensuring that no data duplications occur, storage space can be conserved, thus the load on system resources will be reduced.
   - Efficient persistence: through indexing and query optimization, the amount of system power used to persist a database will be reduced, as data retrieval will be quicker and costly queries will be done without, thus also reducing system load. In addition, connections can be grouped and accessed via a central channel in order to aid persistent storage to the database.
   - Load Balancing: by spreading the systems load across time or across resources the load on the system can be distributed, therefore no system resource will be heavily strained. In the case that the limit for a server has been reached, a new instance or so will have to be created in order to handle the number of increasing

requests. On the other hand, if the usage of a server is way below the capacity, the number of instances will have to be reduced.

- Caching: to ensure no duplication or repeated retrieval of frequent objects or queries; a separate module can facilitate caching; thus system resources will not be used up unnecessarily.

2. Architectural Pattern(s):

    - Blackboard Architectural Pattern

        - This pattern functions by providing a framework for a systems that need to integrate varying, large specialized modules. It entails the use of three components, namely a knowledge source, a control as well as a blackboard in order to facilitate the distributed solving of problems in a system wherein there is no defined process to solve these problems. This pattern is in itself scalable, through its distributed architecture, and as such scalability can be easily achieved across a grid of processors.

    - Master-Slave Architecture Pattern

        - This pattern functions by splitting independent tasks over a number of independent processing grid slaves; wherein the slaves do the main work and the master manages the entire process. This allows for parallel processing, wherein each slave simultaneously performs a system operation and the results of each operation are later merged.

- Aspects of the Scalability discussed above can be seen as an illustration in Figure 1 below.

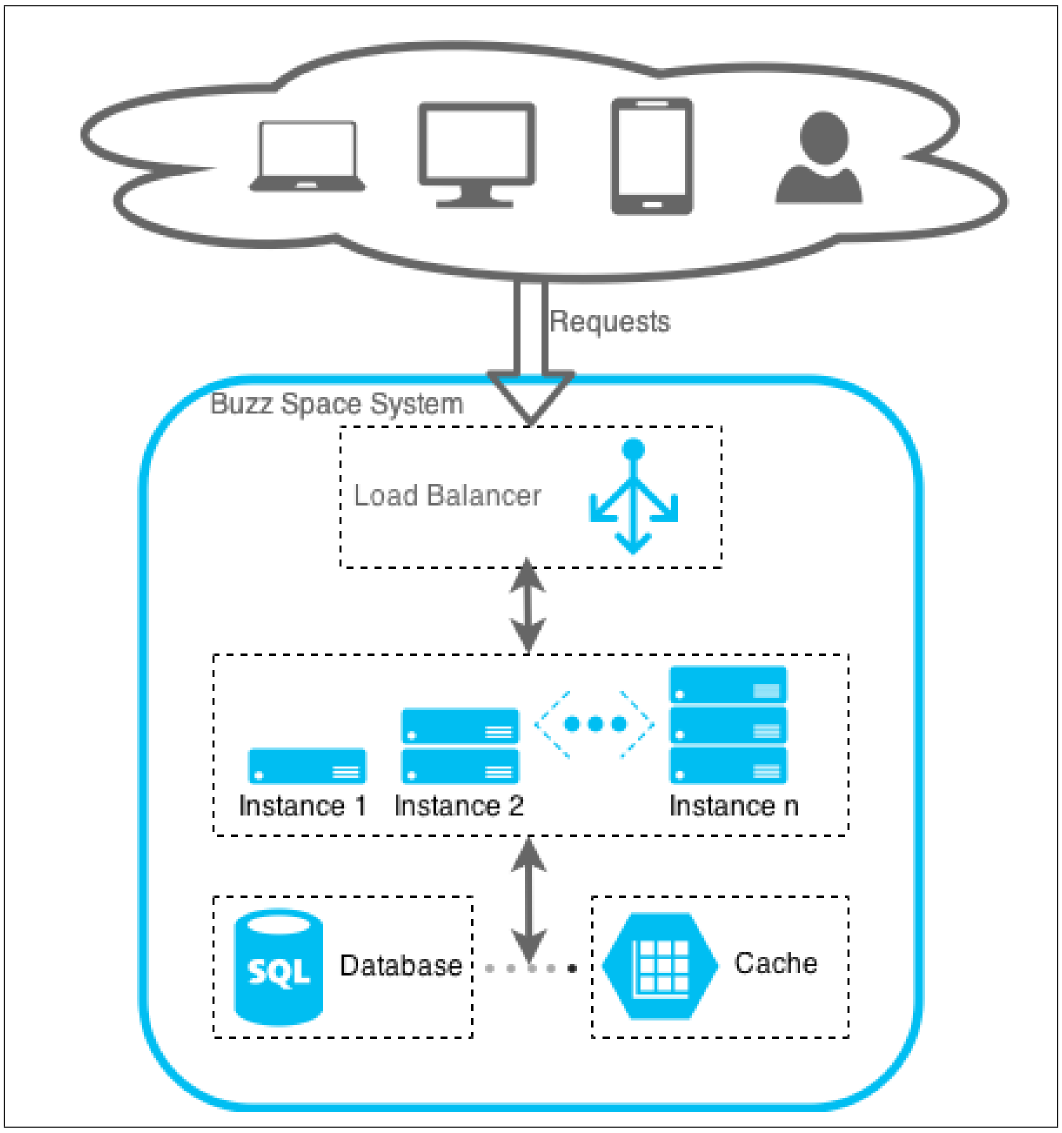

Figure 1: How the system architecture can be set-up to allow many concurrent connections.

2.2.2 Security

Description:

Justification:

Mechanism:

1. Strategy:
2. Architectural Pattern:

2.2.3 Usability

Description:

Justification:

Mechanism:

1. Strategy:
2. Architectural Pattern:

,

2.2.4 Integrability

Description:

Justification:

Mechanism:

1. Strategy:
2. Architectural Pattern:

2.3 Important quality requirements

2.3.1 Performance

Description

Performance is an indication of the responsiveness of a system to execute some action(s) within a specific time interval. This time interval is usually measured in terms of latency or throughput.

> **Latency:** the time it takes the system to respond to any event(s).
>
> **Throughput:** the number of events executed within a given amount of time by the system.

A system with high performance maximizes throughput and minimizes latency.

Justification

We chose performance as an Important quality requirement because the over or under performance of a system will influence other quality attributes of the system. If the system has increased latency and decreased throughput, due to the systems inability to handle increased load (scalability), the performance, responsiveness and usability of the system will suffer.

Strategy

- The system needs to be hardware and software fault tolerant (to a certain degree). The system needs to continue working/running as intended but possibly at a reduced level, rather than breaking/stopping completely.
- The system needs to be responsive. The time it takes for a request to be sent back to a user should be minimized.
- The UI presented to the user needs to be clean, dynamic and use minimal resources. An example would be to use minified JavaScript files and compressed images.
- The performance of the system needs to be optimal, memory or processor intensive tasks should run/execute when there are the least number of active users in order to minimize the impact these tasks will have on performance.

- The system needs to have a coping mechanism when there is a sudden change of environment. (E.g. can handle 100 connections suddenly there is 10000 connections). Performance will suffer if this is not taken into consideration.

- The system should deliver intermediate results or updates to the user when executing a request. For instance, a web page that submits a form via AJAX can have a status/busy indicator to let the user know his/her request is being processed.

- When there is increased database server processing

- In order to keep latency low and throughput high the system needs to cache objects/frequently requested database results This will ensure that frequent objects and queries aren't repeated or fetched over and over again wasting system resources and increasing database server processing.

- An important aspect of performance is to ensure the scalability of the system is optimal. Because an increase in performance directly affects the systems scalability, and in turn the lack of scalability also affects performance.

2.3.2 Plug-ability(Maintainability)

The system as a whole should be designed and developed in such a way that it is modular allowing for additional modules to be added or even removed.

The reason why this is an important quality requirement is because the system should allow the addition of new plug-ins(modules) or the removal of old modules that are no longer required or obsolete. This allows for a more adaptable system as it can be adjusted to a specific users needs. By ensuring plug-ability the system will also be much easier to test. Diagnostics of potential flaws and module clashes can now be eliminated as the system can be tested as individual modules and even as a whole allowing for multiple module configurations.

Mechanisms:

1. (a) There are multiple strategies that can ensure plug-ability one such strategy as mentioned above is to subdivide the system into separate interconnect-able modules or plug-ins.

   (b) Another strategy is to split the system into many services that communicate with one another to perform the required functional requirements.

2. The best Architectural pattern to realize this requirement would be the micro-kernel, this is because this pattern allows for a plug-and-play infrastructure meaning that modules can easily be added or removed or even rolled back to previous versions without affecting other services on the system. This is done by using the Internal servers of the micro-kernel.

2.3.3 Monitor-ability

Description:

Justification:

Mechanism:

1. Strategy:
2. Architectural Pattern:

2.4 Nice to have quality requirements

2.4.1 Reliability and Availability

Description:

Justification:

Mechanism:

1. Strategy:
2. Architectural Pattern:

2.4.2 Testability

Description:

Justification:

Mechanism:

1. Strategy:
2. Architectural Pattern:

2.5 Integration and access channel requirements

2.6 Architectural constraints

3 Architectural patterns or styles

4 Architectural tactics or strategies

5 Use of reference architectures and frameworks

6 Access and integration channels

7 Technologies