Architecture Requirements

Group Name: Group 7_a

Roger Tavares	10167324
Thinus Naude	13019602
Kabelo Kgwete	11247143
Sylvester Mpanganer	11241617
Maphuti Setati	12310043
Ruth Ojo	12042804
Axel Ind	12063178
Lindelo Mapumulo	12002862
Maria Qumalo	29461775

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

Date: 04 March 2015

Contents

1 Access and integration requirements **4**

2 Architectural responsibilities **4**

3 Quality requirements **4**

4 Architecture constraints **10**

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 responsibilitie.
- Quality requirements.
- Architecture constraints as specied by the client.

1 Access and integration requirements

1.1 Access channels

1.1.1 Human access channels

1.1.2 System access channels

1.2 Integration channels

2 Architectural responsibilities

3 Quality requirements

3.1 Scalability

- The Buzz Space discussion board should allow for more than a million concurrent users.[1]
- This scalability will be achieved by running many instances of the application over a cluster of servers or instances. This can be achieved by using a cloud hosting platform such as Amazon Web Services.
- There will need to be a load balancer in order to ensure every request is handled by a server that is not over or at capacity. If the limit for a server is reached (user configurable) a new instance needs to be created to handle the number of increased requests. If the amount of requests are at or below a certain limit (user configurable) the amount of instances needs to be reduced.
- In order to facilitate persistent storage to th database connections need to be grouped and accessed via a central channel.
- Caching of objects/frequently requested database queries will be facilitated by a separate module. The caching will interface with the system as well as the database to ensure frequent objects and queries aren't repeated or fetched over and over again wasting system resources.
- The Scalability discussed here can be seen as an illustration in Figure 1.

[1] Communicated to us via email correspondence with Mrs. Vreda Pieterse

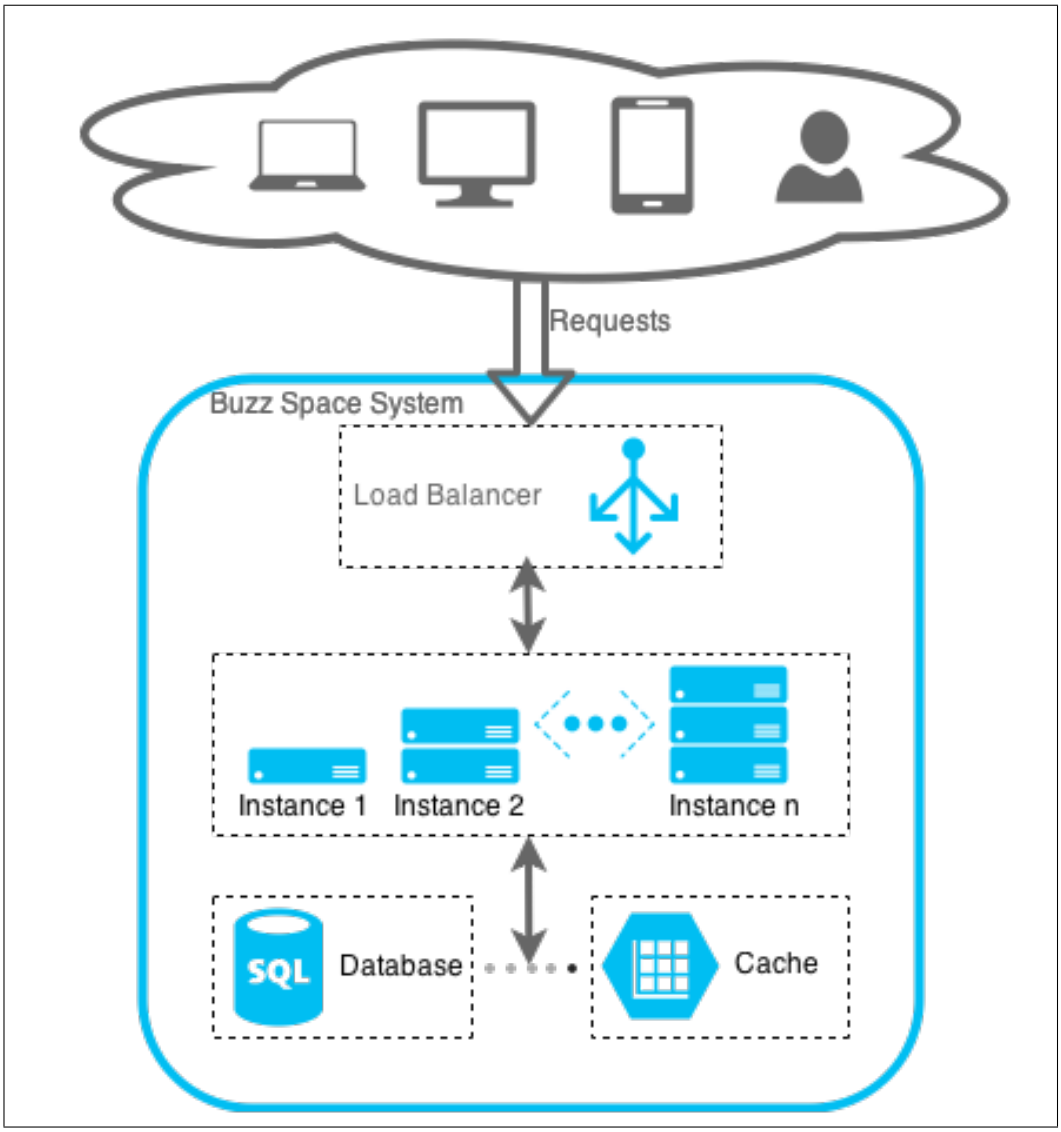

Figure 1: How the system architecture can be set-up to allow many concurrent connections.

3.2 Performance requirements

- 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.

3.3 Maintainability

- Dependency injection will be used to fully decouple the implementation classes. Dependencies are purely on contracts (interfaces) and not on classes.
- The system needs to be capable of improvement in functionality by the addition or replacement of components.
- The system needs to easily facilitate:
  - Detection of problems/errors and their cause(s)
  - Repair or replacement of faulty or deprecated modules without having to replace still working modules.
  - Prevention of total breakdowns.
  - Adaptability to new requirements.
  - Easy deployment of modifications to improve performance or other attributes.

- Throughout the implementation stage strict coding standards/guidelines should be followed. This will keep the code code clean, consistent, and easy to read. See Figure 2 for an example of a suggested code style.

- For easy reference source code documentation will be generated by the Javadoc program.

```
// Type names are camel case.
class NamingExample {
    /**
        * Multiple lines of Javadoc text, wrapped normally.
    * Methods and variables inside a class is named using
    * camel case with first character being lower case.
    * There needs to be a open line before methods.
    * Use tabs instead of spaces.
    */

    // Static finals are to be upper case letters and underscrores
    public static final int A_CONSTANT_VALUE = 5;

    private int[] myArray = {1, 2, 3, 4, 5, 6};// this is fine
    private final int x = 10; // this is fine
    private Color color; // this also fine

    // The first word should be a verb.
    public void doSomething(int a, int b, int c, int d) {
        if(a > b)
            return c;
        return d + 10;
    }

    //This is allowed
    void doNothing() {}

    /**
    * This is a comment
    * @param p1 bla bla bla
    */
    public int method(String p1) { ... }
}
```

Figure 2: Possible style guide for code to be used in the project.

3.4 Reliability and Availability

- The availability of the system needs to be as high as possible if a breakdown occurs, the system needs to recover by itself.

- The system needs to allow multiple or different database connections
- All components of the system need to have redundancies built in.
  - If the database connection fails, a connection to the redundant database needs to be established.
  - If a server instance fails, the error needs to be logged and the server should restart and try to recover from the breakdown.
  - If a data-centre where some of the system instances are hosted goes down the system should only suffer temporarily. The instances need to spread across multiple datacentres or locations.
- The reliability of the system should be tested by using unit tests and acceptability testing.
- The system should allow for a notification to be send to a user, if their request has failed or the delivery there of has failed.

3.5 Security

- The system will interface with LDAP to provide authentication.
- All communication with the database needs to be handled over an encrypted connection.
- The system will need to support encryption (https) via SSL.
- All requests will be logged and if required audited to verify no unauthorised access to the system resources has been gained.
- The system will monitor all service requests in Buzz, irrespective of the channel through which they are requested. These requests will be validated by the "Authorization Interceptor" module which checks whether the user is authorized to use the service that they requested.
- At all times the system needs to minimize the chance of malicious or accidental actions by unauthorised users.
- The system needs to prevent disclosure or loss of information.
- The system needs to be built to withstand common malicious attacks such as SQL injection and cross-site scripting.
- The system needs to be able to detect and mitigate Denial of service (DoS) attacks, either by using a hardware of software approach.

3.6 Monitorability and Auditability

3.7 Testability

3.8 Usability

3.9 Integrability

4 Architecture constraints