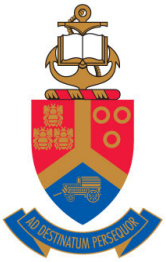

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

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 11, 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):

- Quality requirements.
- Integration and access channel requirements.
- Architectural constraints
- Architectural patterns or styles
- Architectural tactics or strategies
- Use of reference architectures and frameworks
- Access and integration channels
- Technologies

2 Architecture Requirements

2.1 Architectural Scope

This section discusses the boundaries and extent or range of view, outlook, applications, operations or effectiveness the system software architectur needs to address. More spesificaly we will be looking at the following three topics in depth. Persistence, Reporting and Process execution.

2.1.1 Persistence:

Buzz Space needs to be percistant in order to for states to be stored and outlive the many processes done. It will do so by making use of the following:

- Databases, to store and retrieve user accounts data and information.
- Java Data Objects (JDO), a specification of Java object persistence. With it's great transparency feature of the persistence services to the domain model.
- System prevalence, a technique that joins system images and transaction journals to achieve percistancey.

2.1.2 Reporting

Description:

Segregates the system into two applications, where the client makes requests to the server. In many cases, the server is a database with application logic represented as stored procedures.

2.1.3 Layered Architecture

Description:

Partitions the concerns of the application into stacked groups (layers).

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.

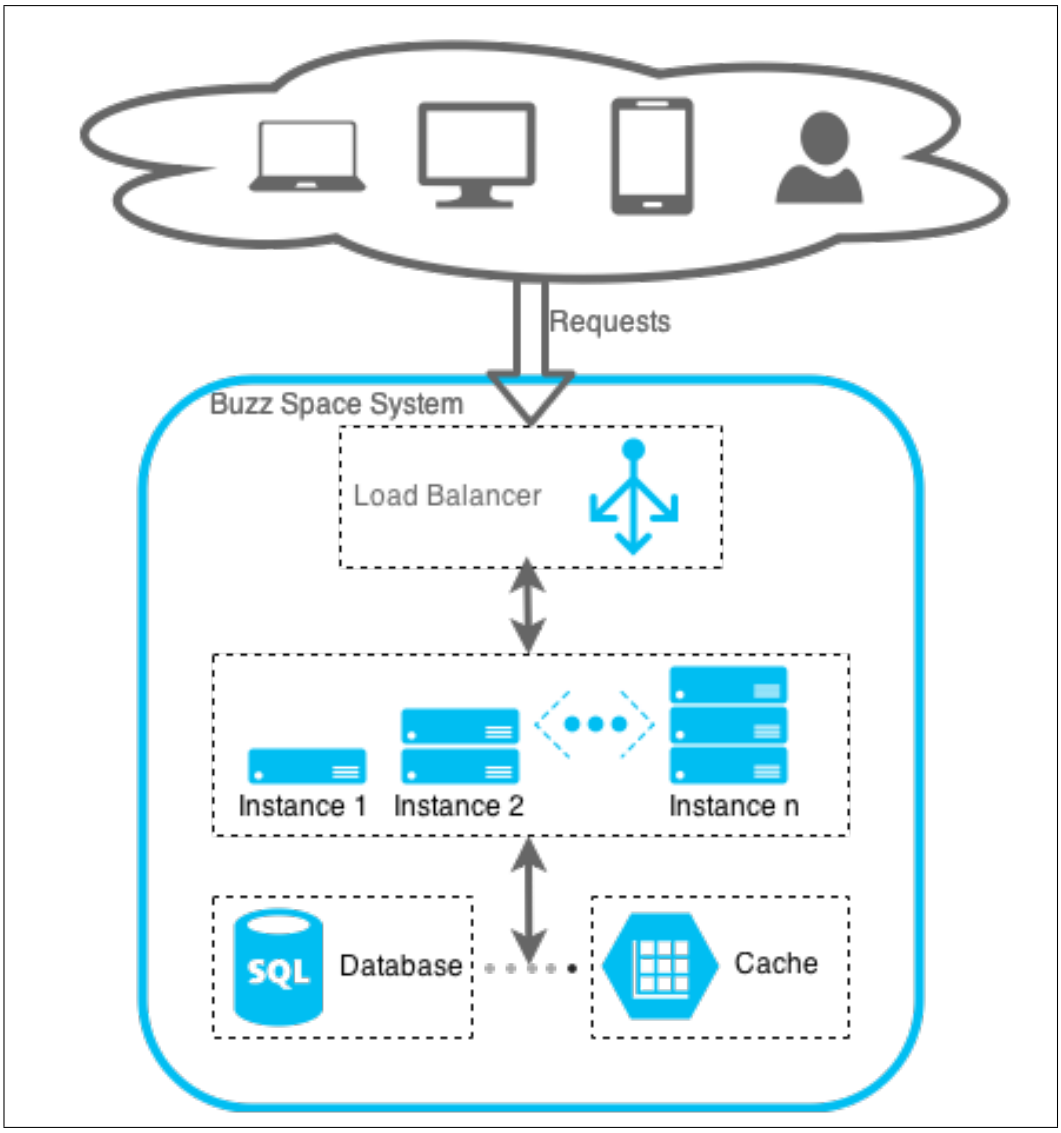

Figure 1: How the system architecture can be set-up to allow many concurrent connections.

2.2.2 Security

Description

Security includes the securing of confidential data stored from unauthorized access and modification. For BuzzSpace, this means:

- Ensuring only genuine users may have access to the systems data.
- Different levels of access should be provided to users.
- Securing vulnerabilities that may exist in the application.

Justification

Security is a crucial aspect of the system. Poor security design and implementation can compromise the data stored by the system and hackers with malicious intentions can succeed to gain access to sensitive information. Using a combination of security strategies and patterns, we can prevent unauthorized users from accessing sensitive information or cause damage to the system.

Mechanism

1. Strategy:

   Security can be achieved by using:

   - Authentication: This is the process that ensures and confirms a users identity.
   - Encryption: This process converts sensitive data into another form, which cannot be easily understood by unauthorized users. Data will be converted to readable format for authorized users.
   - Input Validation: The process of validating all the input to an application before using it. This will prevent malicious attacks such as SQL injections.
   - Auditing and Logging: This includes the recording of the implementation level events and domain-level events that occur when the system is being used. This data will help identify what went wrong if the system experiences malicious attacks.

2. Architectural Pattern(s):

    - Authentication Enforcer pattern
        - This pattern can be used to manage and delegate authentication processes.
    - Secure Base Action pattern
        - Shows centralizing handling of security tasks such as input validation in a base action class.
    - Secure Logger pattern
        - This pattern can be used to log sensitive data and ensuring tamper-proof storage.
    - Secure Session Manager
        - This pattern can be used to show securely centralizing session information handling.

2.2.3 Usability

Description: Buzz should be easy to use and the user should be able to remember how Buzz works; despite their long absence. The interface should appear easy to use; the visuals should not frustrate the user.

Justification:Usability is critical because Buzz is user-oriented and requires that users don't get frustrated with the interface.

Mechanism:

1. Strategy:

    - Descriptive headings that will not be ambiguous to the user.
    - Clear and descriptive "help" messages that will initiate the user into Buzz.
    - Descriptive error messages that will tell the user an error has occurred and the necessary steps to address the errors.
    - Grouped content; this will make the interface look easier to use. If a user is looking for content, they should find related content in the same place. Navigation becomes easier.
    - A user should be given feedback on their actions. If a page is still loading, for example, the user should be aware that the page is loading.

2. Architectural Pattern:

,

2.2.4 Integrability

Description:

The Buzz system should be integrated with ease on any host site.

Justification:

The Buzz system is primarily designed for the CS website, but if in future a need to host it on another site arise,it needs to be easily integrated to the the site without any extra cost and the need to change the implementation of the system.

Mechanism:

1. Strategy:
   The contracts based decoupling strategy can be used to implement integrability and achieve extensibility on the system by providing an interface between the system and the host site which will improves decoupling on the host site and the system.

2. Architectural Pattern:
   Messaging can be used to enhance decoupling and increase flexibility, the Buzz system or host site can change without affecting the other party, both parties will act as independent component that use an interface between them to request services and share resources.

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. This is ideally what we want for the BuzzSystem

Justification

We chose performance as an important quality requirement because the over/under performance of a system will influence other quality attributes of the system. If the system has increased latency and decreased throughput(for example due to the systems inability to handle increased load, or subpar scalability) the performance, responsiveness and usability of the system will suffer. Because the Buzz System will be used by users we want them to use a system the performs optimally and responds to their event(s) with minimal latency.

Mechanism

1. Strategy: The system needs to be responsive. The time it takes for a request to be sent back to a user should be minimized.
    - 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. A system that is off-line has 0 throughput.
    - The UI presented to the user needs to be clean, dynamic and use minimal resources. For example using minified JavaScript files and compressed images. In order to reduce latency.

- 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. For example run batch plagiarism and netetiquette checks at 2AM in the morning.
- The system needs to have a coping mechanism when there is a sudden change in its environment. For example if the system can currently handle 100 connections and suddenly there is 10000 connections. the system needs to know how to cope. Performance will suffer if this is not taken into consideration. See Scalability for possible solution.
- The system should deliver intermediate results or updates to the user when executing a request. For example, 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 the system needs to keep latency low and throughput high. The system can achieve this by storing frequently requested database results or objects in a cache. This will ensure that frequent objects and queries aren't repeated or fetched over and over, 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. Architectural Pattern(s):

    - Many/all of the patterns discussed in this document could assist in performance.
    - Architectural Patterns that allow for scaling of resources are preferred.

2.3.2 Plug-ability (Maintainability)

Description

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.

Justification

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.

Mechanism

1. Strategy:
    - 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.
    - Another strategy is to split the system into many services that communicate with one another to perform the required functional requirements.
2. Architectural Pattern(s): 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

Audititabily or Monitor-ability is a software review where one or more auditors/monitors who are not members of the software development and organization team conduct "An independent examination of the software product to assess compliance with specifications, standards, contractual agreements, functional requirements and other criteria according to the development specification. Software Monitor-ability and audiability is different from testing or peer reviews because they are done by personnel external to and independent of the software development organization.

Justification

We identify the Buzz Space monitor-ability /auditability to be a quality requirement of importance because of the following reasons.

- The Buzz Space system has to accommodate and host a multitude of users concurrently, thus making it prone to various malfunctions and glitches
- The Space needs to be monitored in real time at all times to ensure relevance of topics and subject matters.
- The rating and tagging functionality need to be fair and accurate
- Sufficient feedback and updates of the Buzz Space state must be provided to the users. Users that create threads or just comment on one.
- General software control and application usage..

Mechanism

1. Strategy:

   Auditability and monitor-ability will be achieved by allowing any third party software auditor or monitor support group reviewing the Buzz Space examining it specifically for the aspects of the functional requirements as given by the client. Information such as the systems state and processes in complaints with the specification given. One such auditor may be from a well-established organisation, for example Oracles PeopleSoft enterprise which is UPs current used application.

2. Pattern/Tools:

   Tools like SMaRT, a workbench for reporting the monitor-ability of Service Level Agreements for software services such as Buzz Space may be used. SMaRT aims to clearly identify the service level the service level commitments established between service requesters and providers. This monitoring infrastructure can be used with mechanical support groups in the form of a SMaRT Workbench Eclipse IDE plug-in for reporting on the monitor-ability of Service Level Agreements.

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

Mechanism

1. Strategy:

    - Black-box examines the functionality of an application without peering into its internal structures or workings. This strategy will be employed when no information is known about a component, module or the system as a whole.
    - White-box tests internal structures or workings of an application. This requires the explicit knowledge of the internal workings of the system (Space Buzz).

2. Pattern:

    - Layering: Simplify testability since high level issues will be separated from low level issues. This level of granularity makes the system to be easily testable on every layer separately.
    - Model View Controller: Space Buzz model will be separate from the view and the controller, hence it will be much simpler to have separate test criteria testing different cases. This separation simplifies the development cycle since the model, view or the controller could be tested independently.

2.5 Architectural Constraints

- **JPQL** was chosen by the client, although Neo4j would have been better.

- **Java EE** is the platform that will be used to implement Buzz, the system is constrained to this platform.

- **JSF**, was chosen by the client and forms part of Java EE.

- **AJAX** the client requested that this technology be used.
  ¡¡¡¡¡¡¡ HEAD

- **HTML** is the only markup language that can be used as specified by the client.

- **The operating system** that will run the Buzz server is Linux, as requested by the client. =======

- **HTML**

- **The operating system** that will run the Buzz server is Linux, as requested by the client. We recommend Debian as the Linux distribution to use because:

  - Very complete.
  - Supportive and active community
  - Multi-arch/kernel support
  - Very stable and allows you your freedom

  ¿¿¿¿¿¿¿ origin/master

3 Architectural Patterns or Styles

3.0.1 Object-Oriented

Description:

A design paradigm based on division of responsibilities for an application or system into individual reusable and self-sufficient objects, each containing the data and the behaviour relevant to the object.

Reason for inclusion:

By implementing the very widely used Object-Oriented design strategy, the program will gain a variety of benefits in terms of the reduced complexity and code volume provided by the abstraction, polymorphism, decoupling and inheritance attributes of the architectural pattern. A system of this complexity would be incredibly hard to coordinate in any system that does not permit object-oriented design. Object-Orientation is a de facto requirement in all but the most high-level systems wherein memory availability is not extremely limited.

Main Benefits

- Understandable
- Reusable
- Extensible
- Testable
- Highly Cohesive

3.0.2 Client/Server

Description:

Segregates the system into two applications, where the client makes requests to the server. In many cases, the server is a database with application logic represented as stored procedures.

Reason for inclusion:

Sensitive and personal data pertaining to the name, emails and other client information must be kept secure. Client/Server architecture provides a far higher degree of security and data integrity. The nature of the Buzz forum system necessitates a means of centralised data access and a system featuring high maintainability. The Client/Server architecture ensures that changes made to the system are immediately visible to all clients.

3.0.3 Component-Based Architecture

Description:

Decomposes application design into reusable functional or logical components that expose well-defined communication interfaces.

Reason for inclusion:

The possible future needs of the system necessitate that individual functionalities be modified and/or extended to meet changing requirements. Moreover, should specific functionality be required by secondary systems or other, independent, projects; the ability to copy some part of the functionality of this system may well save significant costs and time. To achieve these goals of reusability, replaceability, extensibility and encapsulation the Component-Based Architecture style must be appropriately used.

Main Benefits

- Ease of deployment
- Reduced future costs
- Ease of development
- Reusability
- Mitigation of complex concerns

3.0.4 Layered Architecture

Description:

Partitions the concerns of the application into stacked groups (layers).

Reason for inclusion:

Provides abstraction, allowing individual client objects to be varied, interchanged and reused as necessary, and loose coupling the extent that, if necessary, the implementation of the forum could be completely modified without having to manipulate the Student object hierarchy at all (interchangeability). Stratifying the object hierarchy of the Buzz forum system will also allow for a significantly improved development environment, wherein a developer need only worry about the integration of his section with the work of others concerned with the same layer of complexity.

Main Benefits

- Abstraction
- Isolation
- Manageability
- Performance
- Reusability
- Testability

4 Architectural Tactics or Strategies

Scalability and Performance

- Optimize repeated processes.
- Reuse resources and results.
- Reduce contention by replicating frequently used resources.
- Clustering.
- Efficient use of storage.
- Load Balancing.
- Caching.
- Use REST (Representational State Transfer) to make BuzzSpace into a scalable web service.

Security

- Authenticate users by requesting username and password when interaction with the system begins.
- Authorize users checking if a user has the rights to access and modify either data or services.
- Encryption to maintain confidentiality of data.
- Input Validation to detect malicious attacks.
- Auditing and logging for identifying and recovering from attacks.

Usability

- Usability is enhanced by giving the user feedback as to what the system is doing.
- Descriptive Error messages must be provide along with the necessary steps to address the errors.
- System must respond to actions performed by the user.
- Separate the user interface from the rest of the application using Model-View-Controller.

Integrability

- Use modular programming to modularize the system.
- Use REST (Representational State Transfer) to decouple BuzzSpace from other software that may need its services.

Maintainability

- Use modular programming to modularize the system.
- Use object-oriented programming to sub divide the sub-system features.

Monitor-ability

- Monitor-ability can be enhanced by using fault detection tactics:
    1. Ping/echo: One component issues a ping and expects to receive back an echo, within a predefined time, from the component under scrutiny.
    2. Heartbeat: One component emits a message periodically and another component listens for it. If the heartbeat fails, the originating component is assumed to have failed and a fault correction component is notified.
    3. Exceptions: These are raised when an anomaly in a component occurs, encounter an exception when a fault is detected.

Reliability

- Reliability can be enhanced by using fault recovery and preventions tactics:
    1. Active redundancy: All redundant components respond to events concurrently. All redundant components will have the same state. When a fault occurs in the responding component, the system will switch to the next redundant component, minimizing downtime.
    2. Checkpoint/rollback: the states of the components will be recorded periodically or in response to certain events. When a fault occurs, the system should be restored to the previously consistent state or checkpoint.
    3. Transactions: The system should bundle several sequential steps in such a way that the entire bundle can be undone at once.

Testability

- Enhanced testability by recording the information that enters the system and using it as input into the test harness, and recording the output of the system components.
- Separating the interface from the implementation to enable substitution of implementations for various testing purposes.
- Creating a specialized testing interfaces to capture variable values to a system component and also seeing the output of the component in order to detect faults.
- The components can maintain useful information regarding its execution internally and then be viewed in the testing interface.

5 Use of Reference Architectures and Frameworks

A reference architecture provides a template solution for an architecture for a particular domain. We will use as our reference architecture the Java Platform, Enterprise Edition (Java EE) and Model View Controller framework for the Buzz Space system. Java EE includes an API and runtime environment for developing and running *large-scale, multi-tiered, scalable, reliable, and secure network applications*.

Java EE will be used to implement a tiered architectural structure, each tier or layer containing one ore more of the Buzz Space modules in conjunction with Java EE modules.

A multi-tiered application is thus an application where *the functionality of the application is separated into isolated functional areas, called tiers*. An example of this can be seen in figure 2 on page 24.

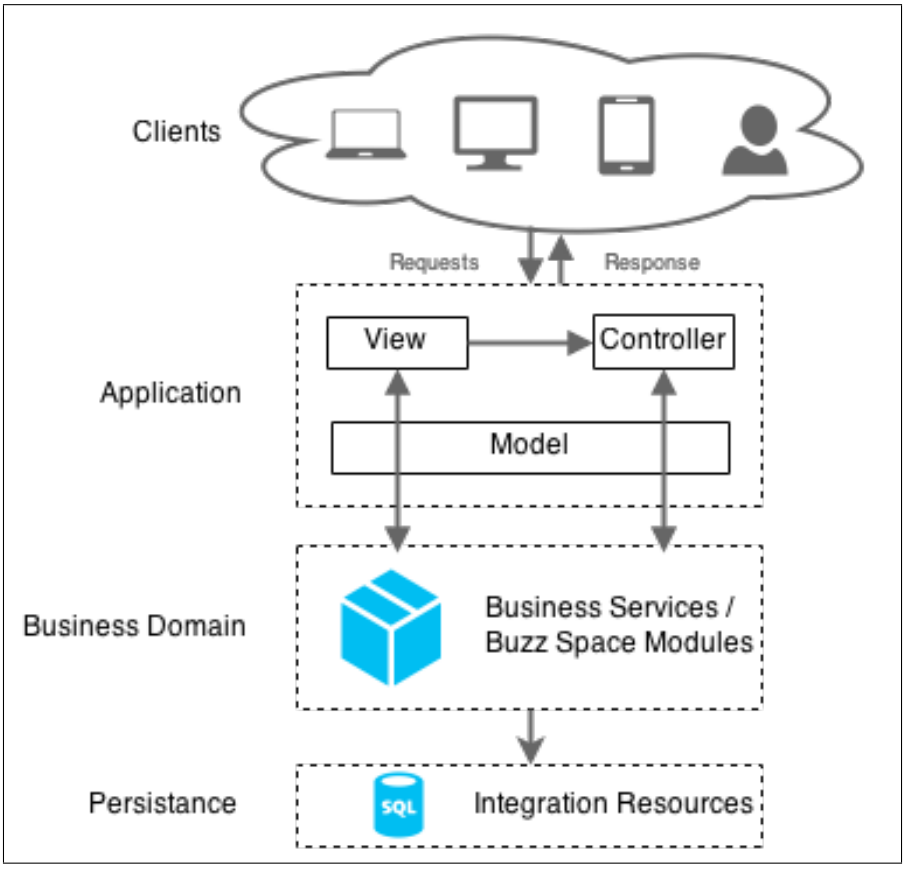

Figure 2: A basic overview of the buzz space tiered/layered application.

6 Access and Integration Channels

6.1 Access Channels

6.1.1 Human Access Channels

Human Access Channels are all the differnt ways a Human being may interact with and access the Buzz Space.

1. The Desktop Computer:
    - This will be the most common way for the human (user) to access the Buzz Space on campus. Either at the Information Technoloy labs or the Library.
2. Laptop:
    - A more portable device for the user to access the Buzz Space is with a laptop.
    - A disadvantage with using ones laptop is that the user will first have to connect to the wifi available to access the Buzz Space.
3. Mobile Phone:
    - Better mobility and portabiliy as the laptop. The user may either connect to the wifi or use his own data to access.
4. Tablet:
    - The last possible human-access-channel is a tablet. It is the same as the mobile phone, the only difference is the screen size of the divice.

6.1.2 System Access Channels

System Access Channels is all the external services or applications that have to interact with the Buzz system in some way or another to ensure that all functionality is available to the users. The main systems that will be used in the current project scope are:

1. The UP Computer Science Website:
    - The Computer Science website must be able to integrate with the system allowing students to access it from module pages providing these users with help and support in their studies.

- This system will eventually replace the current outdated CS forum that is being used.

2. LDAP:

   - LDAP is the database used by the University to store and record all of the students information. This database can be modified or expanded to hold user ranks and post information.
   - This means that access and permissions to the database by the Buzz system need to be provided by the database administrators.

3. The Internet:

   - Since this system will be integrated with the CS website as mentioned above it will in turn also require access to the Internet allowing students to not only access the system locally but also at home or remotely as mentioned in Human Access Channels.

4. External Plagiarism Checker:

   - Developing a plagiarism checker from scratch is time consuming and also outside the scope of the system so in order to assess the posts for potential copyright infringement access to an external checker will need to be incorporated.
   - TurnItIn is a company that provides such a service.

6.2 Integration Channels

Channels

- The Buzz system will access the CS MySQL database to retrieve information of a particular module/course.
- The Buzz system will access the CS LDAP server to gain access to and retrieve details of students, lectures and also a module's enrollment list.

Protocols

The Buzz system will mainly use the HTTPS, LDAP and SMTP protocols.

- HTTPS: This protocol will be use for security, to ensure that a secure connection is maintained and users information is safe.

- LDAP: The protocol that will be used for the integration of the system with the CS LDAP server.
- SMTP: Notifications for plagiarism or posts not conforming to the netiquette rules will use this protocol for the Buzz email system to enable the communication between the administrator (lecture) and students.

7 Technologies

7.1 Platform

- Java EE

7.2 Frameworks

- Java Server Faces

7.3 Operating System

- Linux

7.4 Databases

7.4.1 Relational Databases

- MySQL Database

7.5 Object Relational Mappers

- JPQL

7.6 Languages

7.6.1 Programming Languages

- JavaScript
- Java

7.6.2 Mark-up Languages

- HTML

7.7 Application Servers

- GlassFish Server
- Tomcat

7.8 Dependency Management

- Apache Maven

7.9 Web Services

- SOAP-based

7.10 APIs

- Java Persistence API
- JAX-RS RESTful web services
- JAX-WS web service endpoints
- Java Persistence API entities
- The Java Database Connectivity API (JDBC)
- The Java Persistence API
- The Java EE Connector Architecture
- The Java Transaction API (JTA)

7.11 Others

- AJAX
- Servlets
- Java Server Pages
- JavaServer Faces
- JavaServer Faces Facelets
- Enterprise JavaBeans (enterprise bean) components
- Java EE managed beans

8 Recommended Technologies

8.1 Databases

8.1.1 Object Relational Databases

- Postgresql

8.1.2 NoSQL Databases

- Neo4j
- MongoDB

8.2 Object Data Mappers

To cater for the use of NoSQL Databases

- Hibernate OGM