NORTH-WEST UNIVERSITY
YUNIBESITI YA BOKONE-BOPHIRIMA
NOORDWES-UNIVERSITEIT

Synthesis and evaluation of a data management system for machine-to-machine communication

by

Pieter Willem Jordaan

A dissertation submitted to the Faculty of Engineering in partial fulfilment of the requirement for the degree

Master of Engineering

in

Computer and Electronic Engineering

at the

North-West University

Study Leader: Prof. J.E.W. Holm

April 2013

Soli Deo Gloria

"For the glory of God alone"

Acknowledgements

It is with great reverence and adoration that I thank my Lord for bestowing favour and grace unto me to finish this research project.

A number of persons have been instrumental during this research project and I wish to acknowledge them here:

Jeanne-mari Jordaan

Prof. J.E.W. Holm

Kobus Jordaan jnr.

Maria Ackerman

Kobus Jordaan snr.

Abstract

A use case for a *data management system for machine-to-machine communication* was defined. A centralized system for managing data flow and storage is required for machines to securely communicate with other machines.

Embedded devices are typical endpoints that must be serviced by this system and the system must, therefore, be *easy to use*. These systems have to *bill* the *data usage* of the machines that make use of its services.

Data management systems are subject to variable load and must therefore be able to *scale* dynamically on demand in order to service endpoints. For robustness of such an online-service it must be *highly available*.

By following design science research as the research methodology, *cloud-based computing* was investigated as a target deployment for such a data management system in this research project. An implementation of a cloud-based system was *synthesised, evaluated and tested*, and shown to be valid for this use case. Empirical testing and a practical field test validated the proposal.

Keywords: *data management system, cloud-based computing, machine-to-machine, NoSQL, MongoDB*

Opsomming

'n Gevallestudie vir 'n *databestuurstelsel vir masjien-tot-masjien kommunikasie* is gedefinieer. 'n Gesentraliseerde stelsel vir die bestuur van die berging en vloei van data is 'n vereiste vir masjiene om veilig met mekaar te kommunikeer.

Ingebedde toestelle is tipiese eindpunte wat gediens word deur die stelsel en die stelsel moet dus *maklik bruikbaar* wees. Díe tipe stelsels moet *rekening* hou van *dataverbruik* van die toestelle wat gebruik maak van die stelsel se dienste.

Databestuur stelsels is onderhewig aan veranderlike las en moet dus dinamies kan *skaleer* volgens die aanvraag van die eindpunte wat gediens moet word. Om robuust te wees moet sulke aanlyndiense *hoogs beskikbaar* wees.

Deur ontwerpwetenskapsnavorsing as navorsingsmetodologie te gebruik, is *wolkgebaseerde berekening* voorgestel as die teiken ontplooïngsmetode vir databestuur stelsels in díe navorsings projek. 'n Implementering van 'n wolkgebaseerde stelsel is *gesintetiseer, geëvalueer en getoets*, en validasie daarvan vir die gevallestudie is aangetoon. Empiriese toetse en 'n praktiese veldtoets het die voorstel gevalideer.

Sleutelwoorde: *databestuurstelsel, wolkgebaseerde berekening, masjien-tot-masjien, NoSQL, MongoDB*

Contents

List of Figures

List of Tables

Listings

List of Abbreviations

ACID	Atomicity, Consistency, Isolation, Durability
API	Application Programming Interface
ASIO	Asynchronous Input/Output
AWS	Amazon Web Services
BASE	Basic Availability, Soft state, Eventual consistency
BSON	Binary JSON
CRUD	Create, Remove, Update, Delete
DBMS	Database Management System
DNS	Domain Name Server
EBS	Elastic Block Storage
EC2	Elastic Compute Cloud
ECU	Elastic Compute Unit
ELB	Elastic Load Balancer
HA	Highly Available
IaaS	Infrastructure as a Service
IETF	Internet Engineering Task Force
JSON	JavaScript Object Notation
NoSQL	Not only SQL
OID	Object ID

ORDBMS	Object-Relational DBMS
PaaS	Platform as a Service
QoS	Quality of Service
RDBMS	Relational DBMS
RTT	Round-trip Time
SaaS	Software as a Service
SLA	Service Level Agreement
SQL	Structured Query Language
SSL	Secure Sockets Layer
TLS	Transport Layer Security
UEC	Ubuntu Enterprise Cloud

Any intelligent fool can make things bigger and more complex... It takes a touch of genius - and a lot of courage to move in the opposite direction.

Albert Einstein

Chapter 1

Introduction

In this chapter the problem statement and aim for the project are discussed. The objectives and scope of the project are defined. To conclude this chapter, the research methodology that this project follows, is given.

1.1 Overview

The research theme, namely to synthesize and evaluate a cloud-based machine-to-machine communication and billing system, originated from an actual need in the physical security industry. Such a machine-to-machine communication and billing system basically comprises a centralized server-based data management system as well as machines (units in the field) that communicate via the central data management system.

Instead of just developing a data management system, the question arose whether a data management system should be hosted on a private or a public cloud-based environment. This question was not easily answered, and research followed to investigate the feasibility of such a system on a public cloud.

Directed research requires (i) a *practical* component (the "real-world" problem) and (ii) a *research* component (the "academic" or research problem). The research problem should be derived from the real-world problem and should address a problem that is, in a sense, theoretical in nature (that is, one is allowed to make certain assumptions that define the problem environment as "controlled" in order to be scientific in nature).

Since a data management system is an actual need, a design science research approach was followed. In line with the above discussion, the purpose of this research is thus twofold: *(i) to evaluate the functional capability and performance of a cloud-based data management system in a "controlled" environment, and (ii) to synthesize (create) an artefact that represents a real-world system.*

Design science research is an outcome based research methodology in information systems and computer engineering. [1] The contribution that follows from design science research requires the following [1]:

- Identification of a relevant problem;
- Demonstration that an adequate solution does not exist in the public domain;
- Synthesis of a novel artefact;
- Rigorous evaluation of the artefact;
- Communication of the added value from the artefact;
- Dissemination of the research output.

Validation and verification ("V-and-V") of the artefact was achieved through an empirical testing procedure. Functional tests and performance tests, individually, provide verification of the research, while all tests combined, with a field test, provide validation of this research.

In order to encapsulate the artefact (a data management system), a *use case* was defined that required a system for machines to communicate with other machines in a physical security context. Data management refers to the logistical functionality to manage incoming and outgoing data. This management functionality includes the storage, retrieval and forwarding of data.

Machines in this use case may refer to an exhaustive list of devices, but for this research project refer to embedded network-capable devices, mobile telephones and network-capable computers that are called endpoints. The use case's specific functional requirements are summarized below:

- *Authentication*: the process whereby two machines can securely identify each other;

- *Session support*: the ability to minimize data resends during failed communication attempts by resuming a session, or in case of extreme failures, cleaning a session;
- *Guaranteed delivery of data packets*: a data packet destined for a machine will eventually be delivered as long as the machine is in use - that is, present in the network;
- *Segmented packet support*: large packets can exhaust an embedded system's memory resources and must, therefore, be able to send and receive packets in segments;
- *Persistent storage of data*: the ability to query past events for audit purposes and increased reliability;
- *Packet forwarding*: to enable machines to communicate via a central system;
- *Quality-of-service based data billing*: machines can be billed according to data usage per volume depending on a pre-arranged data rate, allowing low priority machines to operate at lower costs.

The above functional requirements are subject to the following design constraints in order to make the system practically feasible as listed below:

- *Ease of use*: the ability of low-end embedded systems to make use of the data management system. Ideally it should be simplistic to interface with the system;
- *Scalability*: a metric which describes the system's ability to service a highly variable load. If the load increases, the system should scale its capacity accordingly. Conversely, if the load diminishes, the system should decommission unused resources;
- *Availability*: the ability of a system to provide service without interruption - this is important for physical security systems;
- *Security*: provides confidentiality and integrity of messages between machines.

Figure 1.1 shows a typical use case scenario. Remote devices connect to a centralized server to exchange data with its consumers, administrators, and peers. The data is stored on the centralized server for in-time and future delivery of data. Data traffic is also monitored and billing information is generated. Consumers (users) in this figure represent 24-hour security monitoring personnel. The devices represent security devices that are capable of transmitting multimedia security data.

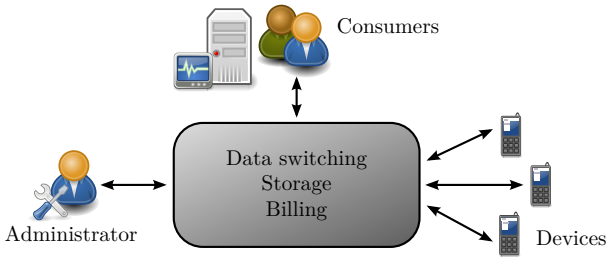


Figure 1.1: Problem illustration

1.2 Background

Embedded systems are often equipped with internet capable peripherals in order to make use of online services. Several applications exist that require online services to act as data management systems. These services are billed according to upstream and downstream data and often the type of data it contains. Furthermore, quality of service (QoS) can be billed according to a service level agreement (SLA). [2]

The primary goal for these systems is to support the variable usage that mobile end-users have. Especially with the advent of the mobile smart-phone and other mobile thin-clients, the need for dynamic scalability has become the focus for many developers. [3]

1.2.1 Current Systems

Currently, there is no publicly available integrated system that provides data management services and billing capability, as defined in this research.

There exists, however, various e-billing and e-management systems for remote energy billing. These systems are exclusively for energy meter billing and are therefore not suitable for data traffic billing. [4, 5] There are other billing systems not mentioned here, but to our knowledge these systems make use of proprietary protocols and architectures that have not been published.

Mobile telecommunication networks employ advanced billing strategies to bill their clients. Clients are billed according to different billing models depending on their need. QoS based billing has become a popular model, but data is not stored during mobile data communication and is therefore not suitable for a data management system. [6]

Great strides have been made towards the development of content-aware internet traffic measurement and analysis. These methods can be used for content-aware billing, but do not provide a means to store the data. [7]

1.2.2 High-level System Architecture

Figure 1.2 shows the high-level system architecture of the system under evaluation. This system provides an established platform that enables multiple clients to manage their data through a simple interface (for example an API[1]). The system provides a gateway that redirects inbound traffic to the storage

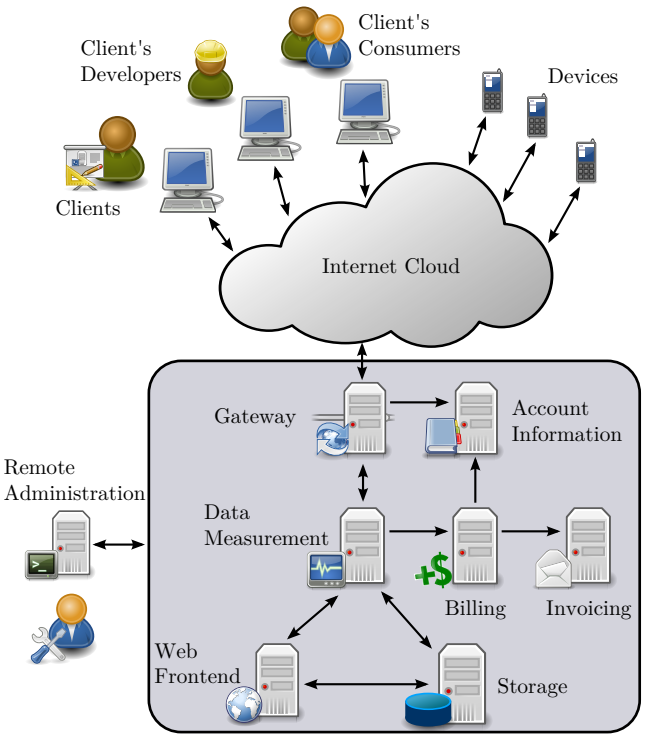


Figure 1.2: Conceptual architecture

[1] An application programming interface (API) provides a standard set of methods by which a developer can interface with a service.

module. Upon establishing a connection, authentication must take place to ensure security. All information, concerning the clients and their registered services, is stored in an account database.

Inbound and outbound traffic is monitored and analyzed for specific packets. Measurements of data payloads (not application specific data) are posted to the billing module that processes prices per volume or other custom pricing rules. Invoices are then periodically created that can be delivered to clients in any form.

An external administrative computer can access the internal network to perform maintenance and administrative tasks remotely.

Recently, cloud-computing has taken flight with public and private clouds easily available. Cloud-computing requires software to be *designed to fail*. Although this may sound like a contradiction, the cloud application paradigm follows good software engineering principles that lead to its remarkable characteristics.

A list of notable characteristics of cloud-computing follows:

- High reliability;
- Versatility;
- Dynamic and infinite[2] scalability;
- On demand service;
- Very low cost;
- High availability;
- Optimal resource handling.

It is obvious that the benefits are not to be ignored for this use case. Applying this kind of development to an application does not limit it to a cloud environment. A cloud application can, in fact, run in *any* environment, but by developing *for* the cloud, the application can effortlessly be deployed into a cloud environment. [8, 9]

It is proposed in this research project that cloud-based design principles should be used to implement a data management system for machine-to-machine communication.

[2]Virtually infinite in a horizontal scaling sense, given the constraints of the hosting environment.

1.3 Research Problem Statement

In the context of design science research, a real-world problem is usually defined and researched. Associated with the real-world problem, again in the context of directed research, is a theoretical (or research) problem that is typically addressed by following an engineering-scientific methodology. Such a methodology is found in design science research.

The ***real-world problem*** was defined and is stated as follows:

Can a data management system for machine-to-machine communication effectively function on a cloud-based platform?

1.3.1 Hypothesis

The ***theoretical research problem*** was defined as an hypothesis that was formulated and tested. The hypothesis is stated as follows:

A cloud-based implementation can provide the functional capability and performance characteristics for a data management system.

1.3.2 Primary Objective

The primary goal was to to synthesize and evaluate a cloud-based machine-to-machine system. Synthesis was done by following engineering principles, and evaluation was done by following engineering-scientific principles.

1.3.3 Secondary Objectives

In order to address the primary objective, a list of secondary objectives were derived, as follows:

- Research, identification and selection of development environment;
- Deployment method identification and selection;
- Research, identification and selection of database(s);
- Protocol synthesis and evaluation;
- Software synthesis and evaluation.

Each of the above objectives were addressed in this research project.

1.3.4 Research Project Scope

The research project was subject to constraints that are listed below:

- The cloud environment must be the primary deployment target;
- The project must be product-hardened so that it can be validated in the field;
- Software must follow the cloud development paradigm - this is *very* important;
- Upstream and downstream data must be monitored for billing purposes;
- Billing reports must be generated;
- Profiling data[3] must be generated and logged;
- The target operating system must be a **nix*[4] variant;
- The requirements in section 1.1 must be met.

The above listed constraints were used to guide the artefact design.

1.4 Research Methodology

Figure 1.3 shows the research methodology that was followed during this research project.

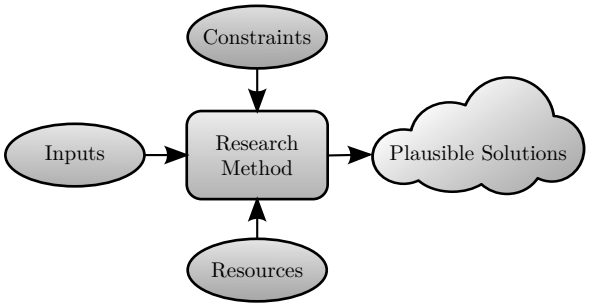


Figure 1.3: Research methodology

[3]Data throughput, access times, resource usage, uptime, etc.
[4]Unix, BSD and Linux

The research methodology that was followed used the process of induction to verify and validate the output of the research method. If the input, constraints, resources and research method are proven (or demonstrated) to be correct, then the output into the plausible solution space *must* be valid. Each of the elements of the research methodology is further discussed in the sections that follow.

1.4.1 Inputs

The inputs to the research were derived from the ***real-world problem***. Real-world requirements were used to define a theoretical research problem and the required functional capability of an artefact.

1.4.2 Constraints

The constraints must be realistic and feasible. Constraints for this research project are listed below:

- Design constraints:
  - Ease of use;
  - Scalability;
  - Availability;
  - Security.
- Current technology limitations.

1.4.3 Resources

Resources are utilized by the research method, which include, but are not limited to, the following:

- Engineering best practices;
- Literature study;
- Measurement tools;
- Development environments;
- Development kits;
- Laboratory environments;
- Development computer.

1.4.4 Research Process Methodology

The design science research method was followed in this research project. An important aspect of the design science research method is *defining the research problem commensurate with the real-world problem.* The research method followed is illustrated in figure 1.4.

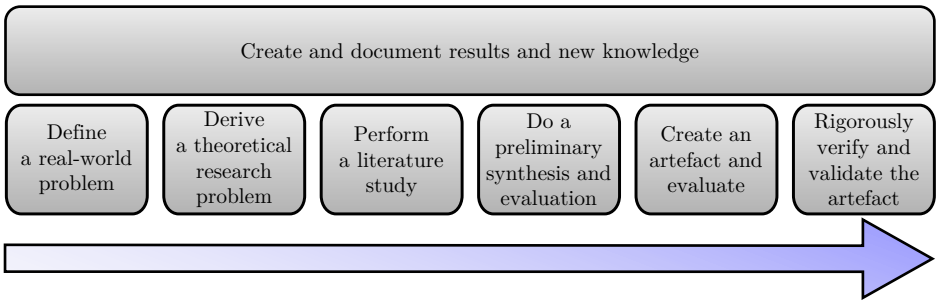


Figure 1.4: Research method

As shown in figure 1.4, the definition of the real-world problem and the derivation of the research problem are followed by a literature study on relevant topics. This, in turn, is followed by a preliminary synthesis which entails the definition of a preliminary artefact for evaluation purposes. A final artefact architecture is selected from the evaluation and is then designed and implemented in detail. The final artefact is verified by means of functional tests and performance tests. Final validation is achieved, again in the spirit of design science research, when all verification tests have shown success and by demonstrating added value in practice by means of practical implementation.

Throughout the research process, a rapid prototyping life-cycle model was followed. This model allowed the development to rapidly converge towards a final artefact. [10]

1.5 Contribution to Research

This section summarizes the contributions made to this research project in order to create, verify, validate and document the artefact. These contributions were spread over a period of two years. A list of contributions is shown below:

- Identification of a ***real-world problem***, namely to provide a data management system for machine-to-machine communication;

- Deriving a ***research problem*** and relevant requirements and constraints, namely to investigate the practical feasibility of a cloud-based data management system for machine-to-machine communication;
- Performing a literature study on relevant topics as listed below:
  - Databases;
  - Scalable computing;
  - Security.
- Rigorous evaluation of identified technologies from the literature study as listed below:
  - Relational databases;
  - Non-relational databases;
  - Grid computing;
  - Cloud computing;
  - Load balancing;
  - Authentication;
  - Encryption.
- Definition of a preliminary architecture as shown in Chapter 3;
- Definition of detail elements of the system architecture as shown in Chapter 4;
- Actual software implementation of the system (artefact);
- Actual implementation of an application programming interface (API) in order to test the system;
- Functional capability testing of each element of the system as shown in Chapter 5;
- Performance testing of the system as shown in Chapter 5;
- Physical deployment of the system on an Amazon AWS instance for a field test of an actual physical security system;
- Critical review of test results as shown in Chapter 5;
- Verification and validation of the artefact, as is evident from the evidence in Chapter 5;
- Documentation of the research.

1.6 Summary

This chapter illustrated the use of design science research as a method for synthesis and evaluation of a cloud-based data management system.

Specific functional capability requirements for the data management system were defined, as shown below:

- Authentication;
- Session support;
- Guaranteed delivery of data packets;
- Segmented packet support;
- Persistent storage of data;
- Packet forwarding;
- Quality-of-service based data billing.

The above requirements followed from a defined use case in a physical security context.

In order to make the data management system practically feasible, the following design constraints were identified:

- Ease of use;
- Scalability;
- Availability;
- Security.

The functional requirements derived from the real-world research problem define the research method's input. Constraints and resources define the boundaries of the research project as outlined above.

By validating the inputs, constraints and resources, the validity of the research project's output can be determined by a process of induction. This process of induction was used to validate the hypothesis:

A cloud-based implementation can provide the functional capability and performance characteristics for a data management system.

Any man who reads too much
and uses his own brain too little
falls into lazy habits of thinking.

Albert Einstein

Chapter 2

Literature Study

2.1 Overview

This chapter discusses the topics studied to aid in the synthesis and evaluation phase. Technologies that were used in the research project are discussed here. A reference architecture is provided as a guideline for the literature study. Topics relevant to the following elements were investigated and are reported on in the following sections:

- Databases;
- Scalability in computing;
- Security.

It is important to note that TCP/IP was used as the transport layer for the system.

2.2 Databases

A database is used primarily to store a collection of end-user data and meta-data in a structured fashion. Meta-data describe data relationships. Figure 2.1 shows a Database Management System (DBMS).

A DBMS is the fabric between a user and a database. It is responsible for translating all application requests into complex database operations. It hides the internals of the database from the application.

Databases are stored in a structure called a schema that defines relationships and columns. The most notable advantages of using a DBMS are listed on the following page: [11]

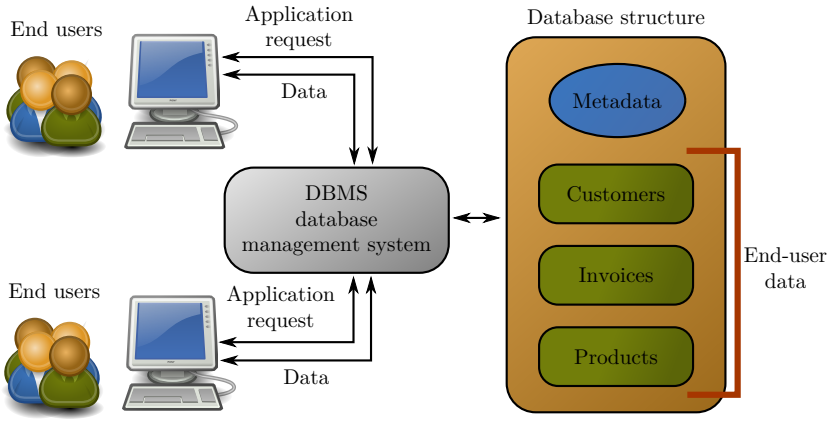


Figure 2.1: Database management system

- Improved data sharing;
- Better data integration;
- Minimized data inconsistency;
- Improved data access;
- Improved decision making;
- Increased end-user productivity.

2.2.1 Database Replication

Data replication is a method to store copies of the data at several independent sites. These sites can be geographically separated to allow for faster access times for local users and also extra security.

Replication is an asynchronous process where a master site replicates to one or more slave sites. All write operations still take place on the master and are then replicated to the slaves. This improves read access times due to redundant sources for reading. Write operations are slightly faster due to read traffic being redirected to the slaves.

One can, for example, use a replicated site for analysis only. This can alleviate the master's work load. Backups can take place on any slave due to the asynchronous operation and without any locking of the master unit. [11]

2.2.2 Database Sharding

In order to scale write operations, one must segment data either horizontally or vertically (by row or by column or both). This allows segments to be distributed (and also replicated) over various sites. Writes are then distributed to the applicable site for the write operation. This can dramatically improve write access times. Read access times are also improved by this method of scaling, unless large queries across all shards or partitions are performed. [12]

2.2.3 Database Cluster

Various DBMS's allow for synchronous replication of data to multiple nodes. Replicated nodes together form what is referred to as a database cluster. All transactions are performed synchronously, which enforces all nodes of a cluster to always be an exact replica of the master at any given time, assuming the master node does not fail during a synchronization attempt.

A cluster is used, almost exclusively, in a local network due to the overhead of synchronous behaviour. A cluster does, however, allow for very fast read access times.

This method of replication can be used in conjunction with asynchronous replication, to form a multi-master circular replication structure, as shown in figure 2.2, which allows for geographic distribution of databases. [12]

2.2.4 Failover Replication

Databases make use of replication to ensure failover support. Master-slave replication allows a slave to be promoted to a master if the master fails. Failover allows transactions to take place with 99.999% (also called the five 9's) availability. This approach, however, requires more hardware resources and server administration. [12]

2.2.5 Database Variants

Various implementations for DBMS's exist, of which the RDBMS and NoSQL databases are the most popular. [13, 14]

2.2.5.1 Relational Database Management Systems

Relational database management systems (RDBMS) have become the *de facto* standard for databases since its introduction in the early 1970's. RDBMS's are popular due to their ACID (Atomicity, Consistency, Isolation, Durability) transactional properties.

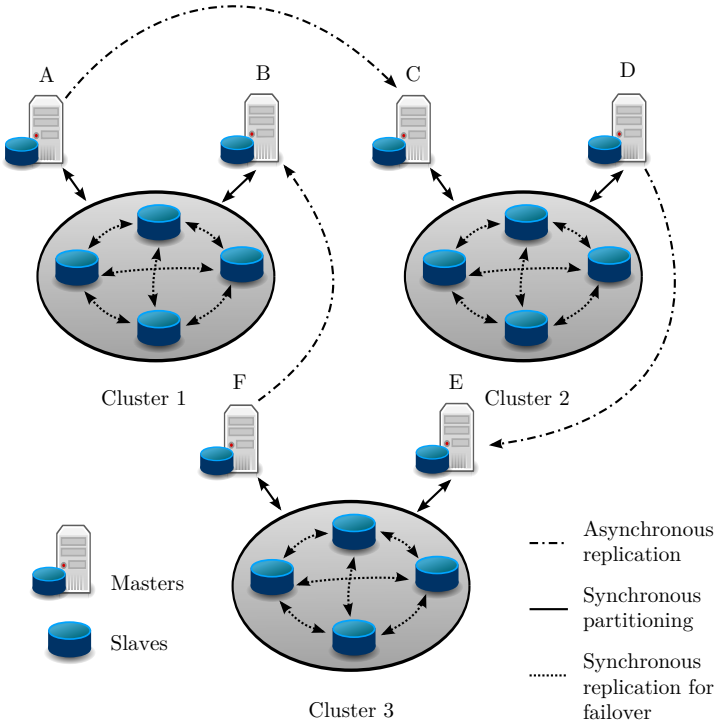


Figure 2.2: Circular geographic replication

RDBMS's are mainly accessed by means of a language called *structured query language* (SQL pronounced officially as "es-queue-el" and not "se-quel"). An example query using SQL is given in listing 2.1. This simple query creates a table in a preselected database and adds columns `P_Id`, `LastName`, `FirstName`, `Address` and `City`. A primary key `pk_PersonID` is also defined. [11]

```
CREATE TABLE Persons (
    P_Id int NOT NULL,
    LastName varchar(255) NOT NULL,
    FirstName varchar(255),
    Address varchar(255),
    City varchar(255),
    CONSTRAINT pk_PersonID PRIMARY KEY (P_Id,LastName)
);
```

Listing 2.1: SQL example

RDBMS's are usually more apt for small, but frequent, read/write transactions, and large batch read transactions. They generally do not function well for the intensive workloads of large scale web services like Google, Amazon, Facebook and Yahoo. [13]

Many RDBMS implementations exist, both open-source and commercial, each with different features and levels of support.

2.2.5.1.1 Microsoft SQL Server SQL Server 2008 R2 is Microsoft's flagship database. Development tools ship with SQL Server that greatly reduce development and debugging times. SQL Server integrates effortlessly with other Microsoft technologies such as Excel, Windows Server, Sharepoint and Visual Studio. Windows is the only possible target environment, however. [15]

SQL Server 2008 R2 Parallel Data Warehouse extend the base features of SQL Server 2008 by allowing shard-like capability. [16]

SQL Server 2008 R2 Datacenter allows optimal resource usage by deploying to virtual environments. It makes use of *Hyper-V* technology to boost performance of its virtual databases. [17]

2.2.5.1.2 Oracle Database Oracle's Oracle 11g Database is a commercial database that is rich in features and support. Oracle 11g has an advanced scalable infrastructure through their Real Application Clusters (RAC). An abbreviated list is given that shows some of Oracle 11g's notable features:

- Partitioning;
- Replication;
- Cluster capability;
- Cross platform;
- Failover support.

[18, 19, 20]

2.2.5.1.3 PostgreSQL PostgreSQL is an open-source object-relational database management system (ORDBMS). PostgreSQL implements most of the SQL standard set of features and has extended it by adding the features shown below:

- Data types;

- Functions;
- Operators
- Aggregate functions;
- Index methods;
- Procedural languages.

ORDBMS's offer the advantage of complex data, data type inheritance and object behaviour. This allows for sophisticated schemas that aren't possible without dramatic workarounds in standard RDBMS's. [21]

2.2.5.1.4 MySQL MySQL is an open-source RDBMS that has been commercialized by Oracle. The commercial MySQL editions offered by Oracle include advanced support and tools but does not offer additional functionality or performance for the database itself. MySQL is unique in that it offers different database engines, each with different properties. These engines serve as solutions for different use cases. MySQL is available on virtually any platform, including embedded environments.

MySQL offers a cluster edition that supports multi-master replication. Inside a cluster, automatic partitioning and sharding take place. Failover is automatically implemented through a special management process that is part of the cluster.

Each cluster comprises two data nodes that are grouped together, a management process and an optional MySQL server front-end. This group acts as a failover for a specific set of shards and it is proven that an availability of 99.999% is possible. Inside a cluster, it is possible to add new data nodes. These nodes will automatically be utilized to balance the cluster's data load. [12]

2.2.5.2 NoSQL

With the growing scale of web service users, traditional RDBMS's do not perform well. NoSQL (not only SQL) systems have taken flight, due to the lack of scalability in RDBMS's. NoSQL implementations are categorized under the following three main database types:

- Wide-column store;
- Document store;
- Key-value store.

Each NoSQL variant has different advantages, drawbacks and limitations. NoSQL databases are generally faster than relational systems and are inherently scalable, but mostly lacking in ACID compliance. Most NoSQL systems are rather BASE (Basic Availability, Soft state, Eventual consistency) compliant. [13]

Three different systems were investigated, as discussed in the sections that follow, namely:

- Cassandra - a wide-column store;
- MongoDB - a document store;
- Memcached - a key-value store.

2.2.5.2.1 Cassandra Cassandra was first developed by Facebook, before being open-sourced in 2008. Apache made Cassandra a top level project and is constantly improving it. Cassandra is a wide-column store, which implies that data is stored in columns (a tuple). Column families are stored separately in a file and contain one or more columns, which are analogous to tables in a relational system. Column families are contained within a row. Super columns are also possible, which implies that a column field may contain any number of other columns. Listing 2.2 shows a JSON[1] representation of a possible data set. [22]

Cassandra features a single-node system, where each node in a cluster is essentially the same. In order for replication and sharding to function, each node should know of at least one other node, which enables all nodes to eventually know of every other node. Data is automatically distributed among new nodes with the option of replication for failover. [22]

Cassandra does not make use of trees to store data, but rather stores data sequentially on a disk. This reduces random disk I/O that, in turn, delivers better write performance but slightly slower read performance than other systems. In terms of indexing, Cassandra does not support secondary indexes for super columns, which implies that data must be de-normalized. A work-around exists for this shortcoming, by implementing reverse-indexes [22, 23]

Cassandra delivers many client API's, but it works mainly through Thrift's remote procedure call system. [22]

[1]JavaScript Object Notation. `http://www.json.org`

```
UserContact = { // Keyspace
    name: "user profile",
    Pieter Jordaan: { // Column family
       pieterAddress: { // Row Key
           name: "pieterAddress",
           value: { // Super column
                city: {name: "city", value: "Potchefstroom"},
                street: {name: "street", value: "555 Hoffman Street"},
                zip: {name: "postalcode", value: "2531"}
           }
       },
       kobusAddress: {
           name: "kobusAddress",
           value: {
                city: ... ,
                street: ...,
                zip: ...
           }
       }
    },
    John Doe: {
       ...
    }
}
```

Listing 2.2: Cassandra data model

2.2.5.2.2 MongoDB

10gen's[2] MongoDB (from hu**mongo**us) is a commercially supported open-source NoSQL document store. It is developed in C++ for optimal performance and is available for most platforms. It delivers client API's for most of the popular programming languages and also defines a wire protocol.

MongoDB is document-oriented that easily maps to programming language data types. Traditional table joins are exchanged for embedded documents that dramatically improve performance. Documents are schemaless and can be dynamically changed. High performance and scalability are achieved by single-document-only transactions.

MongoDB supports indexing in embedded documents and arrays to further increase performance. Write latency can be greatly reduced by making use of MongoDB's streaming writes.

[2] http://www.10gen.com

High availability is achieved by replication with automatic master failover. Scalability is attained through automatic sharding and shard balancing. MongoDB has a query router (`mongos` process) that automatically routes queries to the appropriate shards.

The data model MongoDB uses is shown in the form of a list below:

- A MongoDB system contains a set of databases;
- Each database contains a set of collections;
- A collection consists of a set of documents;
- Documents are a set of fields;
- A field is a key-value pair;
- A key is a name (string);
- A value can be any BSON[3] object.

MongoDB features a rich query language based on JSON objects. A simple query that matches all documents with the name "John Doe" is shown in listing 2.3. Queries may also contain regular expression searches. Listing 2.4 shows a C++ client example program that creates a "John Doe" object and requests it. Each document by default receives a globally unique ID called an OID (Object ID) that can be used to locate a specific document from a default index created on the `\_id*` field.

```
1 {name: {first: 'John', last: 'Doe'}}
```

Listing 2.3: Simple MongoDB query

[3]Binary version of JSON. `http://bsonspec.org`

```cpp
#include <iostream>
#include <client/dbclient.h>
using namespace mongo;
using namespace std;
void run() {
    // Connect to MongoDB
    DBClientConnection c;
    c.connect("localhost");
    // Create a BSON object
    BSONObj p = BSON( "name" << "Joe" << "age" << 33 );
    // Insert
    c.insert("tutorial.persons", p);
    // Query
    BSONObj q = c.findOne("tutorial.persons", QUERY( "age" << 33 ) );
    cout << q.getStringField("name") << endl;
}
int main() {
    try {
        run();
        cout << "connected ok" << endl;
    } catch( DBException& e ) {
        cout << "caught " << e.what() << endl;
    }
    return 0;
}
```

Listing 2.4: Simple MongoDB C++ example

MongoDB clusters consist of shard servers, config servers and routers. Each shard contains a replica set (each node runs a `mongod` MongoDB process). Config servers are started by the same `mongod` process. The routers are special `mongos` processes. A typical MongoDB deployment is shown in figure 2.3.

Documents, at the time of this writing, have a 16Mb size limit, but for applications where large or many files are to be stored, MongoDB's GridFS is a solution. GridFS is a MongoDB based distributed file system that allows for the storage of files of virtually any size.

Map/reduce is a function that can aggregate data across multiple shards concurrently. The mapping function is applied to all matching data and emits only the required fields. The emitted data is then grouped together according to a key field and is passed to a reduce function. Reducing the

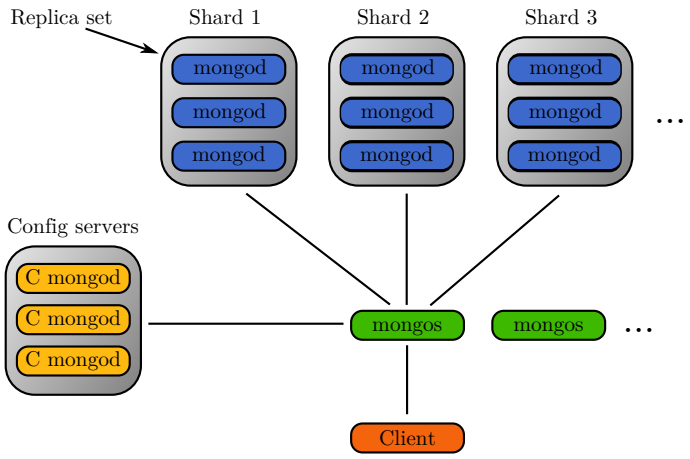


Figure 2.3: Typical MongoDB cluster [24]

data implies that the mapped (grouped) data is manipulated in some way to provide a single element per key field. The reduction function's output should be in the same format as its input, due to the fact that the reduction process can be run multiple times.

Map/reduce can be used, for instance, to calculate an average of some field or fields. The mapper emits the field to be averaged for each matching document, a counter field that is set to 1, and the key field that is grouped on. The reduction function sums the field per key field and accumulates the counter field. A finalize function can be called on the reduced data to take the total and divide it by the count per key value. This provides a total, count and average field grouped by the key field.

MongoDB provides a flexible map/reduce function that can be used to aggregate and manipulate data. The output of such operations can also be merged or even further reduced to allow for incremental aggregation.

Client drivers are supplied for popular programming languages such as Java, C++ and python among others. The client drivers provide the following capabilities:

- Connection pool;
- Queries;
- Replica set queries[4];
- Inserts;

[4]These queries are only sent to replica sets.

- Batch inserts;
- Find and modify operations;
- Consistency level options.

MongoDB has been successfully applied to the following use cases:

- Archiving and event logging;
- Content management systems;
- E-Commerce;
- Gaming;
- Mobile;
- Data store;
- Agile development environments;
- Real-time statistics and analytics.

[24]

2.2.5.2.3 Memcached A key-value store, like the open source `memcached`, is used to improve data retrieval rates by using an in-memory cache. When a key's value is required, the cache is checked first for the key. If the key exists in the cache, the data is retrieved from memory. Alternatively the data is retrieved from the persistent database (that is slower) and is then stored in the cache. When that same key is queried again, it can be retrieved from the fast cache rather than the slow database. For data updates and deletions the same principle applies. The higher the cache hit-ratio, the faster the data retrieval rate is. [25]

Many popular web-services make use of `memcached` as stated on its website[5] at the time of writing:

- Twitter;
- YouTube;
- Flickr;

[5] `http://memcached.org`

- Wikipedia;
- WordPress;
- Digg.

2.2.6 Selection of Database Management System

As the system will store data persistently, it is important to select a database technology to address this requirement. Two technologies were surveyed and evaluated: (i) relational databases, and (ii) non-relational (NoSQL) databases.

Relational databases were found to be well-suited for small, but frequent, read/write operations, but do not scale well for large-scale deployments due to their transactional characteristic. In order to allow for infinite scalability of the system, a NoSQL database was selected.

The key-value store `memcached` does not provide any means of scaling and cannot represent complex data, and was therefore not applicable for this use case.

Cassandra and MongoDB were both found to be applicable to the data management system use case, as both can scale infinitely and both have flexible data models. Cassandra, however, does not support secondary indexing of fields and was therefore found not as suitable for the database management system. MongoDB was thus selected as the DBMS for this system due to its features, as listed below:

- Inherently scalable;
- Inherently redundant;
- Single rich document design;
- Secondary indexing;
- No referential integrity;
- Powerful aggregation capability;
- Easy deployment;
- Flexible schema.

2.3 Scalable Computing

Systems can scale-up (vertical scaling) by using faster and larger resources. Scale-up has an upper limit that is due to technology limitations. When applications require more resources, a scale-out (horizontal scaling) approach must be taken. [19]

Although this research focuses on cloud computing, it was necessary to review alternatives for the sake of completeness. Two popular scale-out techniques are grid computing and cloud computing and are discussed in the sections below. [26]

2.3.1 Grid Computing

Grid computing is analogous to electrical grids. Wall outlets allow linking to an infrastructure of resources. The resources are generated, distributed and billed according to use. Where the power plant is located and how the power is distributed is of no consequence to the user. Grid computing acts as a fabric connecting disparate resources across a network in order to function as a virtual whole. The goal of grid computing is to provide on-demand resources for users.

Software that divides workload into fragments, to be distributed, is compulsory in order to function in a grid environment. Grid computing is usually used for scientific applications. [27]

Grids usually process batch-scheduled operations where a local resource manager manages resources for a grid site. Users submit batch jobs to the grid system. An example batch for a user could be:

1. Stage input data from a URL to local storage;
2. Run application for 60 minutes on 100 processors;
3. Stage output data to a remote FTP server.

From this batch, it is obvious that there is no user interaction and that the grid must wait until 100 processors are available for 60 minutes. [26]

2.3.2 Cloud Computing

Cloud computing, in contrast to grid computing, is designed to provide services rather than resources:

- Platform as a service (PaaS);

- Software as a service (SaaS);
- Infrastructure as a service (IaaS).

These services are deployable across a large pool of computing and/or storage resources, accessible through standard protocols. Each service can be scaled up or down dynamically depending on application needs.

Clouds employ virtualization techniques to create an abstraction and encapsulation layer. Analogous to threads on a multi-core processor, user applications are deployed on many virtual machines (called instances) in order to scale an application.

As instances are loosely coupled, the need for integration with other techniques is necessary to provide high availability and failover support. Also, collaboration between instances for distributed computing requires middleware to negotiate tasks. [26]

2.3.2.1 Amazon AWS

Amazon provides a public cloud service named EC2 (Elastic Compute Cloud) as part of Amazon Web Services (AWS) that offers affordable and flexible cloud solutions. A per-usage billing model is followed by AWS, that allows for flexible scaling with minimal costs.

A variety of EC2 instances are provided by AWS and table 2.1 compares some of the basic instance types. CPU power is measured in terms of ECU's (Elastic Compute Unit) where one ECU is the equivalent of a 1.0-1.2 GHz 2007 Opteron or 2007 Xeon processor. EC2 instances can use elastic block-storage (EBS) devices to increase storage space.

Table 2.1: EC2 instance types

Instance	Cores	ECU's	RAM	Architecture	I/O	Storage
Micro	1	Up to 2[6]	613MB	32/64-bit	Low	None[7]
Small	1	1	1.7GB	32/64-bit	Moderate	160GB
Medium	1	2	3.75GB	32/64-bit	Moderate	410GB
Large	2	4	4.5GB	64-bit	High	850GB
Extra Large	4	8	15GB	64-bit	High	1.69TB

[6]Micro-instances share available resources for short bursts.
[7]Micro instances can only use EBS.

AWS also provides CloudWatch that monitors cloud service metrics and can be configured to scale up or down automatically depending on the system's load.

Another service provided by AWS is the Elastic Load Balancer (ELB). The ELB is used to distribute load to a collection of servers in a round-robin fashion by default, but this behaviour can be customized. ELB's can be incorporated with the CloudWatch system to automatically scale with a greater load. Automatic failover support is also inherent in ELB systems.

If a scalable DNS service is required, one can utilize Amazon's Route 53. Route 53 hosts domains with multiple zones and will automatically scale if the load increases.[28]

2.3.2.2 Eucalyptus

Eucalyptus is an open-source private cloud solution that implements the same API as EC2 and can also integrate with EC2. Ubuntu delivers a free reference architecture for Eucalyptus in the form of Ubuntu Enterprise Cloud (UEC). UEC can be installed on any computer with hardware-virtualization support. [29]

2.3.3 Load Balancing

Load balancers are used to make web services highly-available (HA) and scalable in an unobtrusive (transparent) fashion. Load balancers route incoming traffic to server farms in such a way that the total load is balanced over the farm. This allows for greater loads than a single server could process. Furthermore, failover for a server farm can be implemented on the load balancer. [30] DNS and software load balancing are surveyed.[8]

2.3.3.1 DNS Load Balancing

DNS load balancing uses round-robin load balancing when domain names are requested. A domain name zone can map to more than one public IP in a round-robin fashion. DNS load balancing can also use IP filters to geographically load balance requests depending on the source IP of the request. It is possible to dynamically change DNS entries for failover support on DNS level.[31]

[8]Hardware load balancers are not applicable for a cloud environment.

2.3.3.2 Software Load Balancing

Software load balancers work on the TCP/IP level. Some balancers allow for HTTP analysis in order to better distribute requests for HTTP backends. HAProxy[9] is a popular software load balancer and has successfully been deployed to Amazon EC2.[32]

2.3.4 Selection of Scalable Computing Method

As the hypothesis requires a cloud-based implementation, Amazon's AWS was chosen as the deployment target. This choice, however, does not limit the artefact's capability to run on other platforms. Eucalyptus provides the same API and can therefore be used as a drop-in replacement for private deployments. Amazon provides affordable deployment options, and also the ability to scale on demand depending on the load. The following services are well established on Amazon AWS which allows for easy deployment:

- Load balancing;
- Domain name service;
- CloudWatch service monitoring;
- Dynamic block storage.

Depending on the deployment requirements, any or both of DNS and software load-balancing can be used to distribute load. Due to a load-balancer's unobtrusive nature, the selection of specific load-balancing technology does not directly impact the functionality or empirical performance of the system.

2.4 Security

Security methods address the following aspects usually by employing cryptography:

- Confidentiality - only permitted users may take part in communication;
- Integrity - ensuring the message content is pristine;
- Authentication - establishing credibility of source and destination;
- Non-repudiation - proving origin and integrity of data.

[33]

[9] `http://haproxy.1wt.eu/`

2.4.1 Authentication

In order to trust a sender or receiver, they must be authorized. Cryptography techniques can be used to prove the authenticity of both sender and receiver. User name and password combinations are the most basic form of authentication. Certificates and keys provide a more secure and preferred method of establishing authentication. [33]

Authentication can also be securely established by means of an nonce digest exchange. In principle, a server sends a random string of bits (the nonce, sometimes called a challenge or salt value) to the client and the client responds with the nonce hashed together with a password. The hash function is exclusively a one-way hash function such as SHA-256. This process ensures that man-in-the-middle attacks cannot acquire the client's password. If the server sends a message with the password and same nonce hashed together, the authenticity of the server can also be established. [34, 35]

2.4.2 Cryptography

Encryption (and its counterpart decryption) form the basis of cryptography and is mostly used to ensure confidentiality. Two types of encryption methods exist:

- Symmetric-key;
- Asymmetric-key.

[33]

2.4.2.1 Symmetric-key Cryptography

Symmetric-key cryptography uses a shared key between both sender and receiver of a message. The sender encrypts the message with the key while the receiver decrypts the message with the key. Figure 2.4 shows how symmetric-key cryptography works. Popular symmetric-key algorithms in use are listed below:

- Data encryption standard (DES);
- Advanced encryption standard (AES);
- International data encryption algorithm (IDEA);
- Blowfish;

- CAST-128;
- RC5.

[33]

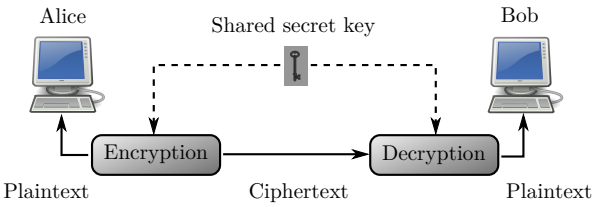


Figure 2.4: Symmetric-key cryptography

2.4.2.2 Asymmetric-key Cryptography

With asymmetric cryptography, a key is shared and used by senders for encryption purposes. This key, however, cannot be used to decrypt encrypted data. That would not be secure as the key is made available to the public. Figure 2.5 shows how asymmetric-key cryptography functions. A private key for Bob is generated and is only available to Bob. This key is the only key able to decrypt ciphers previously encrypted by the public key. In this way an encrypted message can only be decrypted by Bob. Common asymmetric cryptography algorithms are Rivest, Shamir and Adleman (RSA) and Diffie-Hellman. [33]

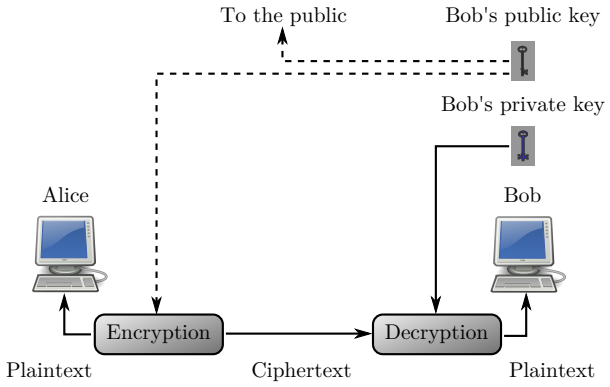


Figure 2.5: Asymmetric-key cryptography

2.4.3 Secure Sockets

Sockets can be secured by utilizing one or both of SSL (Secure Sockets Layer) and TLS (Transport Layer Security). Both methods apply security at the transport layer which implies that any TCP/IP based system can be encapsulated in secure sockets. [33]

2.4.3.1 SSL

SSL provides security and compression for application layer data. SSL is usually used to secure HTTP data, but it can be used for any application layer protocol. SSL provides the following services to applications:

- Fragmentation;
- Compression;
- Message integrity;
- Confidentiality;
- Framing.

[33]

2.4.3.2 TLS

IETF[10] has standardized SSL in the form of TLS, which has deprecated (not replaced) SSL. TLS only differs from SSL in the following aspects:

- Version number;
- Cipher suite;
- Cryptographic secret;
- Alert protocol;
- Handshake algorithm;
- Record protocol;

[33]

[10]Internet Engineering Task Force

2.4.4 Selection of Security Method

Nonce authentication was preferred over standard username/password authentication for improved security. The use of a hashed nonce-value makes the system immune against man-in-the-middle attacks. Furthermore the SHA-256 hash was found to be easily implementable and is not performance intensive.

As TCP/IP was selected as the transport layer, it is fairly simplistic to encapsulate an application protocol in a secure sockets layer. As TLS is the revised standard, it was chosen as an optional encryption layer for the system. Encryption will ensure message integrity and confidentiality of communication. Encryption is however performance intensive for embedded systems.

It is important to note that encryption was optional for increased security and it was not a requirement for the research problem, although it was relevant to the real-world problem.

2.5 Summary

The reference architecture was provided as a guideline for the literature study, and databases, scalability and security technologies were studied.

NoSQL technology was selected due to relational databases being less apt for large-scale deployments. MongoDB was selected as the specific NoSQL DBMS for this system for reasons listed below:

- Inherent scalability;
- Inherent redundancy;
- Single rich document design;
- Secondary indexing;
- No referential integrity;
- Powerful aggregation capability;
- Easy deployment;
- Flexible schema.

As the hypothesis of this research requires cloud-based deployment of the data management system, Amazon AWS was chosen as the target deployment. Amazon AWS provides notable features as listed on the following page:

- Low cost deployment options;
- Automatic scalability depending on load;
- Well established cloud-based services.

In order to distribute load across cloud-based instances, DNS load balancing and software load balancing were identified as distribution methods. Depending on the size of a deployment, either or both of DNS load balancing and software load balancing can be used to distribute load across compute instances.

Nonce authentication was selected as authentication method for the data management system. Nonce authentication was shown to be easily implementable without compromising on security. Immunity against man-in-the-middle attacks was achieved by utilizing nonce authentication.

TLS, being the new standard for secure socket communication, was selected as an optional encryption layer to encapsulate an application protocol. Encryption ensures message integrity and confidentiality during communication.

From the knowledge gained from the study and the technology selections made, a preliminary architecture was defined to aid in the preliminary synthesis and evaluation.

A common mistake that people make when trying to design something completely foolproof is to underestimate the ingenuity of complete fools.

Douglas Adams

Chapter 3

Preliminary Synthesis and Evaluation

3.1 Overview

This chapter provides the preliminary synthesis on the basis of the literature study. A preliminary architecture is provided for the elements that were synthesized and for the system as it stands today. Each element is evaluated on the basis of the stipulated requirements in Chapter 1.

3.2 Preliminary Architecture

Figure 3.1 shows the preliminary architecture of the core elements of the system:

- Data management system (functional unit 1.0);
- Storage system (functional unit 2.0);
- Interface between the data management system and endpoints (interface 1);
- The storage system's interface with the data management system (interface 2).

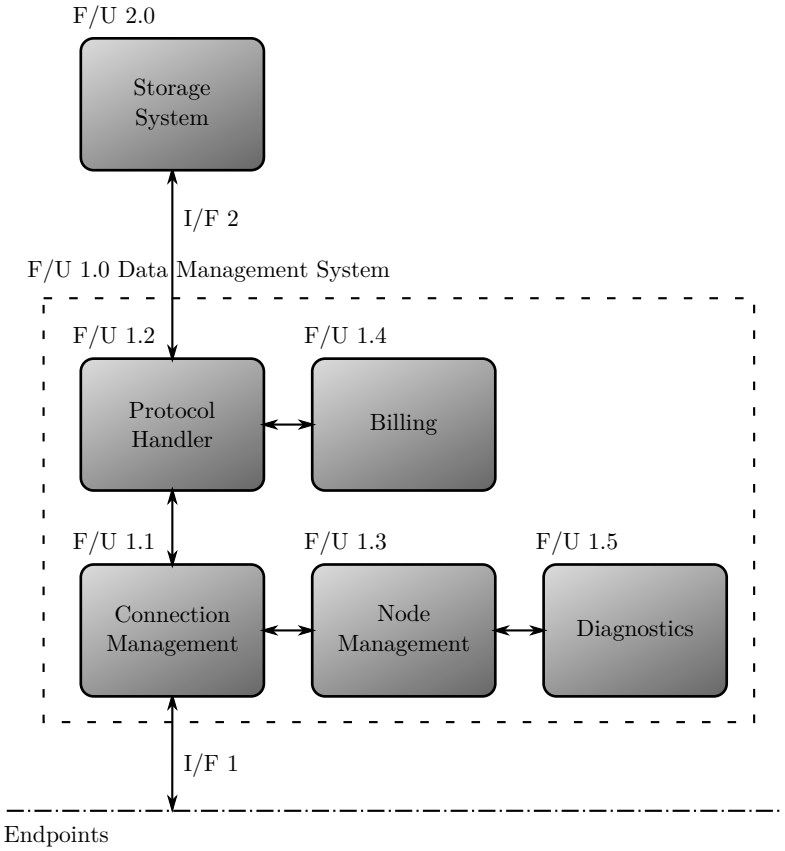


Figure 3.1: Preliminary architecture

3.2.1 Data Management System Protocol

A protocol (I/F 1) defines the communication between the server and endpoints. The protocol is responsible for authenticating endpoints and negotiating a session, which occurs in the handshaking phase. Furthermore it encapsulates endpoint packets in a packet format that allows for validation of the packet's data integrity and provides packets with a session-unique identifier. The identifier per packet allows for acknowledgements of successful packet transmission and also for negotiating progress during session establishment. The protocol functions the same in both directions with the exception of the handshaking phase.

The handshaking phase uses a predetermined password that is only known by the server and the endpoint. The authentication phase uses random data and the password in a hashed form to exchange data between either end. In the event that the password is incorrect, the derived messages will fail. Any man-in-the-middle will not be able to derive the password from communication. In this manner, both the endpoint and server can securely establish a connection.

A few important design requirements must be kept in mind for the protocol. The protocol must have negligible data overhead for efficient transmission of data. Data integrity validation methods must be used that are efficient with respect to processing time and additional data overhead. The handshake phase should allow for declaring a new session or continuing with an interrupted session. Packet segmentation should be possible to accommodate large packets.

Overall, the protocol must be easily implementable on any embedded architecture and should be inexpensive in terms of processing overhead. Figure 3.2 shows the state diagram for a basic client protocol. The server side is similar, with the exception that it listens for multiple connections.

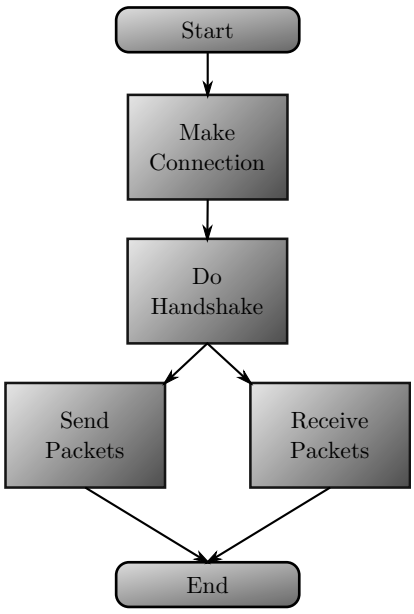


Figure 3.2: Preliminary client protocol state diagram

After successfully receiving a packet, an acknowledgement should be sent back. A packet consists of the following fields irrespective of order, shown on the following page:

- Either a source or destination identifier[1];
- Unique packet ID for the session;
- Optional payload offset[2];
- Payload size;
- Payload data;
- Verification data for payload.

3.2.2 Storage System

The storage system (F/U 2.0) is the central module of a data management system. Packets, endpoints, accounts and billing information are stored, updated and removed here. The storage system must be able to scale horizontally for read and write operations, in order to make the overall system scalable. The storage system should also have a redundancy layer to protect the system from data loss and ensure high availability. Figure 3.3 shows the preliminary architecture for the storage system.

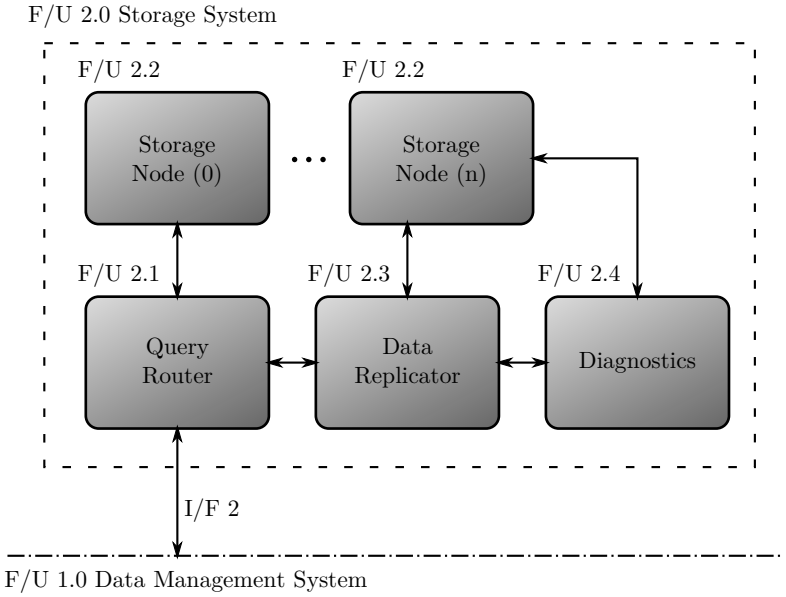


Figure 3.3: Preliminary storage system architecture

[1]Source is for received packets while destination is for sent packets.
[2]In the case of segmented packets

Query Router (F/U 2.1)

The query router has the function of routing requests to the correct storage nodes, which makes database sharding or partitioning possible. All requests pass through the router first, after which the router forwards the query to the correct partition or partitions. Queries include all CRUD operations and also any storage system administration operations.

If the level of consistency of a read is not important, read operations can be forwarded to replicas. By utilizing replicas for read operations, the load on the primary nodes can be alleviated. Furthermore, replicas can be used to perform backups without any downtime of the storage system.

Storage Nodes (F/U 2.2)

The storage nodes are database systems that store data and perform operations on data. There can be multiple storage nodes in both sharded and replicated configurations. The combination of all the shards will be the complete data set. Each shard can have any number of replicas. It should be noted that a storage node will be a typical DBMS.

Data Replicator (F/U 2.3)

Storage nodes can be used as replicas for other storage nodes, providing fail-over support for a shard. The data replicator is responsible for replicating data between the nodes and ensuring the required level of consistency between all the replicas.

Diagnostics (F/U 2.4)

Profiling and status data is generated and stored by the diagnostics module. The profiling and status data can be used to detect problems with storage nodes and find possible bottlenecks in the storage system.

3.2.3 Data Management System

Figure 3.1 shows the data management system (F/U 1.0). The data management system serves as the front-end of the system to the endpoints. Endpoints connect to the system and exchange data with the server via the protocol interface.

The data management system acts as a shared-nothing system, with only the central database as a failure point. The storage system, however, is also a redundant system that has no single point of failure. The system can scale horizontally depending on load and the storage system in a like manner. Each instance of the scaled data management system will be referred to as a node.

Connection Management (F/U 1.1)

A single node can manage multiple concurrent connections from endpoints. It is necessary to be able to close connections on demand for consistent interaction between multiple nodes when units reconnect. F/U 1.1 also monitors connection health and closes idle and broken connections after a predefined period[3]. Connection times and any errors are logged in the storage system.

Each connection polls the storage system for new packets in the endpoint's queue and notifies the protocol handler to process the packet for transmission to that endpoint. Rate limiting for incoming and outgoing packets are also implemented in this functional unit.

Protocol Handler (F/U 1.2)

F/U 1.2 handles the data management system protocol. Every data packet is processed according to the protocol definition. Any irregularities or inconsistencies are not tolerated and will lead to a connection being dropped.

Node Management (F/U 1.3)

Node management refers to the node's ability to manage itself in terms of other nodes. This includes monitoring heartbeats that are also stored on the storage system. If any server fails, all other nodes will be notified within a predefined minimum period and a backup or redundant node can take control of the disconnected endpoints.

Nodes are also able to improve the polling delay for packets by directly notifying the relevant node of new data that it has processed for an endpoint. This leads to an event-driven system that is more efficient than a polling-based system.

[3]This period will ideally be long as persistent connections are preferred.

Billing (F/U 1.4)

The billing function measures the amount of data processed by the protocol handler. The cost for the amount of data is then appended to the account holder's bill according to a service level agreement.

Diagnostics (F/U 1.5)

Similar to the storage system's diagnostic module, the data management system's diagnostic module keeps track of profiling data and any errors.

3.2.4 Database Interface

The database interface (I/F 2) is the component in the system that is the most critical in terms of performance. This interface defines both the protocol between the database management system and the storage system in terms of CRUD operations as well as the actual data definition of objects that will be stored or retrieved. The interface's protocol is directly dependent on the underlying DBMS chosen for the storage system.

The following list shows the basic data definitions required for the system to function:

- Account information:
  - Account holder details;
  - Account QoS details;
- Endpoint information:
  - Reference to account;
  - Endpoint details;
  - Account wide unique ID;
- Connection logs:
  - Connection start and end timestamps;
  - Any error conditions;
  - Reference to endpoint that connected;
- Incoming packet queue:
  - Reference to source endpoint;

  - Session unique packet ID;
  - Payload meta-data[4];
  - Payload data;
  - List of destinations;
  - Status of packet;

- Outgoing packet queue:
  - Reference to destination endpoint;
  - Reference to source endpoint;
  - Session unique packet ID;
  - Payload meta-data;
  - Payload data or a reference to the data[5];
  - Status of packet;

- Billing logs:
  - Reference account;
  - Reference endpoint;
  - Data usage;
  - Price according to QoS;

3.3 Evaluation

The evaluation of the preliminary synthesis was performed by ensuring that all requirements had been addressed (that is, linked) to relevant system functional modules in the preliminary design architecture as obtained from Chapter 1.

3.3.1 Functional Capability

Each of the defined functional capability requirements that were addressed and evaluated is discussed in the following sections.

[4]The packet size and number of segments.
[5]If an incoming packet forwards data it is not necessary to duplicate the data.

3.3.1.1 Authentication

The handshake module of the protocol ensures authentication of both server and endpoint.

3.3.1.2 Session Support

Session support was implemented in the handshake protocol. This provides continuation of previously interrupted sessions and forcing clean sessions in case of terminal failure.

3.3.1.3 Guaranteed Delivery of Packets

With the acknowledgement of packets, delivery can be guaranteed. In the case where no acknowledgement has been received, it is clear that a packet is still not successfully sent. During session negotiation, the progress will reflect which packets have been delivered and can be acknowledged at that stage.

Payload verification is used to guarantee that the payload is uncorrupted. Any failure in the verification process will withhold acknowledgement and force a retransmission.

3.3.1.4 Segmented Packet Support

Together the offset field and packet ID field provide segmented packet support. An offset of 0 declares a new packet, any offset after that, for the same packet ID, will append the segment to the combined payload.

3.3.1.5 Persistent Storage of Data

Due to the storage system being a persistent data store, all packets will be persistently stored and replicated.

3.3.1.6 Packet Forwarding

The destination and source fields in the packet allow for forwarding of packets. When the server parses the identifier in the destination field, the packet is automatically appended to the destination's packet queue.

3.3.1.7 QoS Based Data Billing

The QoS negotiated in the service level agreement indicates the price per data unit. F/U 1.4 utilizes this metric to bill an account accordingly.

3.3.2 Performance Characteristics

In order to evaluate the practical feasibility of the system, the performance characteristics of the preliminary synthesis are discussed. The evaluation was performed on the basis of the performance characteristics defined in Chapter 1.

3.3.2.1 Ease of Use

Low-end embedded systems require interfaces which have minimal impact on its communication, storage, and processing resources. It is for this reason that the protocol should be structured in such a fashion that it does not impact negatively on an endpoint's primary functions.

As the protocol is the only interface that endpoints have with the system, the ease of use is determined mostly by the protocol.

3.3.2.2 Availability

In order for the system to provide uninterrupted service to endpoints, it should be highly available. This implies that redundant nodes should be in place to replace disrupted nodes.

Due to the shared-nothing architecture of the data management system, it is fairly simple to provide fail-over support. In the event of a node failure, a backup node can assume the role of the failed node with no down-time. Availability is also dependent on the storage system's availability. The storage system provides multiple replicas for each shard that in turn provides fail-over support.

3.3.2.3 Scalability

The system can service a large number of endpoints and must therefore be scalable. Furthermore the load can vary, which requires flexible scaling. As load increases, the number of nodes should increase accordingly. As load decreases, the number of nodes should decrease accordingly.

In the same manner that the system is available due to its shared-nothing architecture, it is also scalable. Depending on the system's load, more data management nodes can be started. Likewise, if the storage system's load increases, more shards (and their replicas) can be started. If load decreases, the data management nodes and storage nodes can be retired to free resources.

3.3.2.4 Security

For endpoints to securely communicate, the system must enforce strict security measures.

Security is achieved through the authentication step during the handshake process. Any intruder will not be able to pass the authentication phase. Also, if the server's domain or IP-address is hijacked, the endpoints will not be able to pass the authentication phase.

Data packet integrity checks further improve the security of the system as corrupted data will be rejected.

3.4 Summary

From the preliminary architecture, defined with the aid of the literature study, a preliminary synthesis was performed. System elements were derived and defined from the requirements stipulated in Chapter 1.

From the evaluation in this chapter it is evident that the requirements have been met. A matrix that shows the requirements addressed by each function or interface is shown in table 3.1. From the matrix, it can be seen that all requirements were met by one or more functions or interfaces.

Table 3.1: Requirements vs. architecture elements matrix

	F/U									I/F	
Requirement	1.1	1.2	1.3	1.4	1.5	2.1	2.2	2.3	2.4	1	2
Authentication	x	x	-	-	-	-	-	-	-	x	x
Session support	x	x	x	-	-	-	-	-	-	x	x
Guaranteed delivery	-	x	-	-	-	-	-	-	-	-	x
Segmented packets	-	x	-	-	-	-	-	-	-	x	x
Persistent storage	-	x	-	-	-	x	x	x	-	-	x
Packet Forwarding	-	x	x	-	-	-	-	-	-	x	x
QoS billing	x	x	-	x	-	-	-	-	-	x	x
Ease of use	-	-	-	-	-	-	-	-	-	x	-
Availability	x	-	x	-	x	x	x	x	x	-	x
Scalability	x	-	x	-	x	x	x	x	x	-	x
Security	x	-	-	-	x	-	-	-	x	x	x

All that is not perfect down to the smallest detail is doomed to perish.

Gustav Mahler

Chapter 4

Detail Synthesis and Evaluation

4.1 Overview

This chapter provides the refined preliminary synthesis by giving more detail on the system. The synthesized system is evaluated with reference to the system requirements, as is reported on in this chapter.

4.2 Detail Synthesis

The same architecture from section 3.2, that was shown in figure 3.1, applies to the detail synthesis. The main system was developed in C++, but source code is omitted for brevity.

4.2.1 Data Management System Protocol

The data management system protocol relies on TCP/IP as the transport layer for the system to encapsulate the application protocol in.

It is important to note that the data management system protocol is dependent on (and builds on) the TCP/IP features listed below:

- Guaranteed delivery of per session packets;
- FIFO ordering of packets queued per session;
- Low level checksums for packet integrity;
- Flow control;
- Encapsulation of payload data;

- Standardized protocol;
- Large support base.

[36]

4.2.1.1 Handshake

The handshaking phase of the protocol is used to authenticate the endpoint to the server and also to authenticate the server to the endpoint. It is important to ensure the credibility check succeeds for both the server and the endpoint in order to provide secure data transfer. Figure 4.1 shows the handshake phase's message sequences.

The process is always initiated from the endpoint as the endpoints may not have a predictable public IP address. As soon as a connection is established with the server, the server transmits a random 16-byte string. This string will be referred to as the cryptographic salt or challenge value. The endpoint then sends its handshake packet (shown in figure 4.2 with bit 0 indicating the LSB) to the server. After successfully parsing the packet, the server responds in kind with the packet shown in figure 4.3. All handshake packets are appended with an SHA-256 hash digest. SHA-256 is easily implemented and is secure in that the hashing process cannot be reversed in a reasonable time.

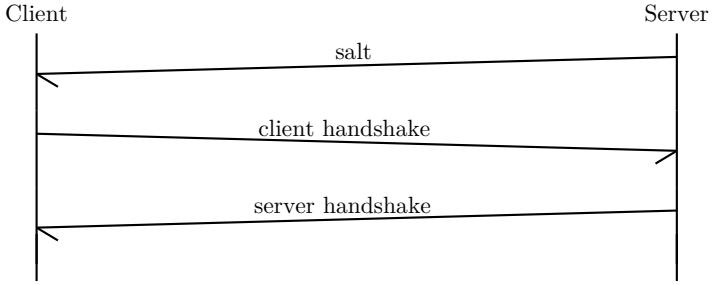


Figure 4.1: Handshake message sequences

Descriptions of the fields in the both the endpoint and server handshake packets are given here:

Version This indicates the version of the protocol. This can be updated with newer releases of the protocol. Any backward compatibility with newer versions is handled by the server side for increased ease of use by endpoints.

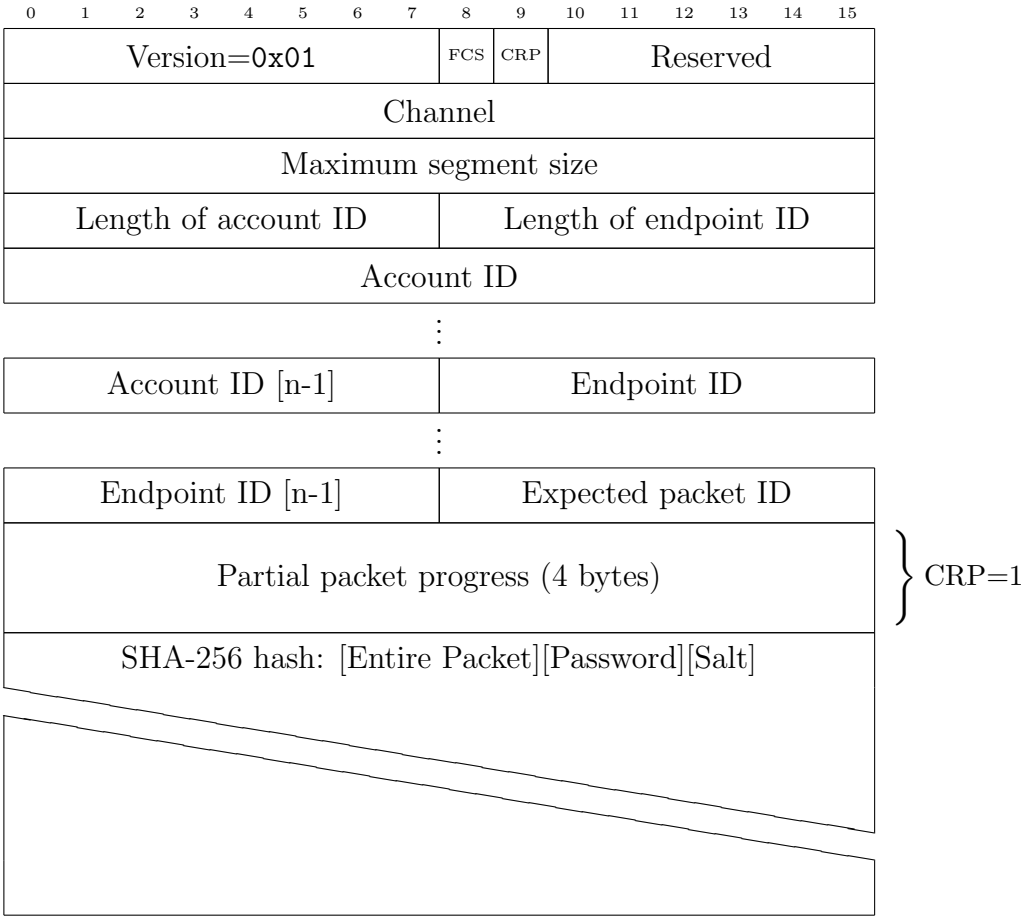


Figure 4.2: Endpoint handshake packet definition

FCS Forces a clean session, which implies all partial packets and unacknowledged packets are flagged as broken. This is used when an endpoint is first started or if either end loses synchronization or detects an error.

CRP Continues receiving a partial packet from a previous session. This is field is mutually exclusive with FCS.

Backoff Backoff defines a period during which the server forces a client to not reattempt connection. After sending its handshake packet, the server will close such a connection. A backoff period is required to prevent inactive or blacklisted endpoints from causing denial of service problems.

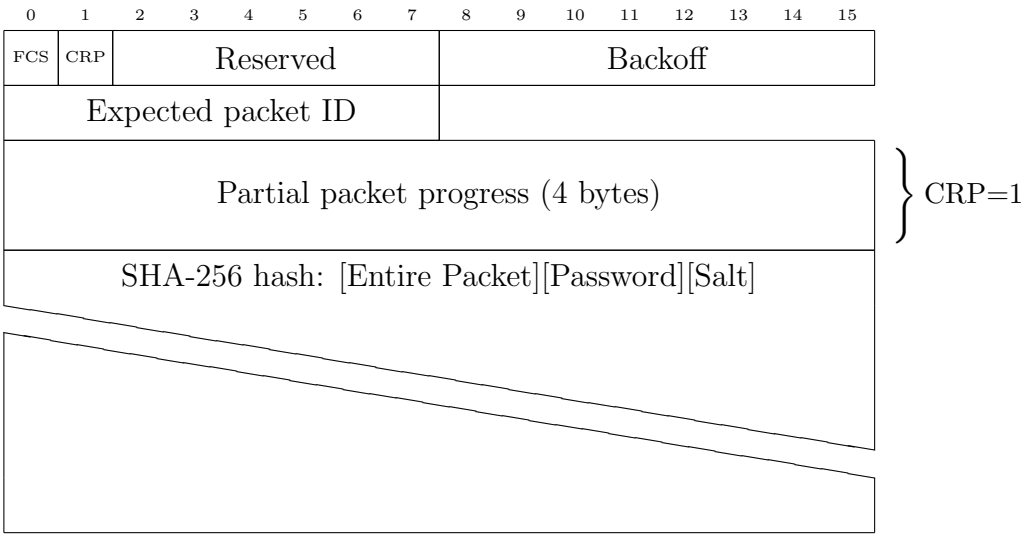


Figure 4.3: Server handshake packet definition

Channel The channel of the endpoint that is attempting to connect. A single endpoint may have multiple physical internet connections that can be used to increase throughput.

Maximum segment size The maximum segment size that the server is allowed to send. By default the server will send large packets in segments of this maximum size in order to avoid additional data overhead.

Length of account ID The number of bytes in the account ID.

Length of endpoint ID The number of bytes in the endpoint ID.

Expected packet ID This is the packet ID that should be used with the next transmission. The expected packet ID is used to acknowledge packets to the other end without resending acknowledgements.

Partial packet progress If the CRP-bit is set, this field contains the progress of the expected packet. Segmented packets can then be resumed from an offset rather than resending all the data.

SHA-256 hash The entire packet (except the hash field itself) is hashed together with the unique password of the endpoint and the session's salt. This has a two-fold use, firstly providing security and authenticity checks and secondly, verifying the integrity of the handshake packets. If the expected hash doesn't match the received hash, it is either due to data corruption or an unauthorized endpoint.

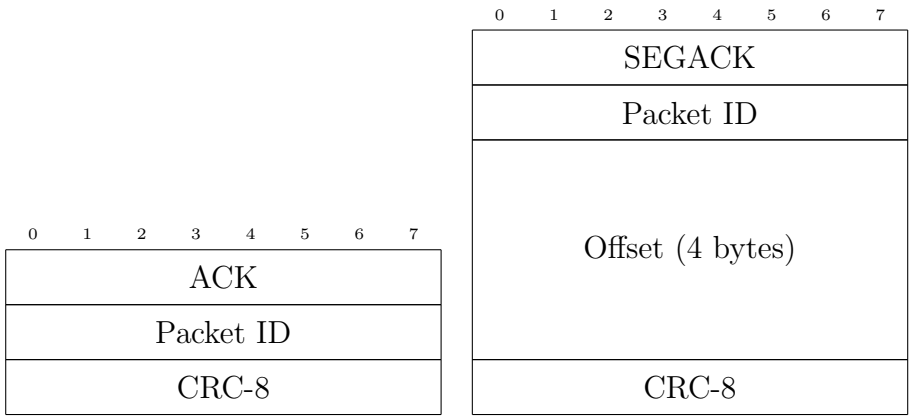
4.2.1.2 Packet Definitions

All protocol packets consist of a header with a CRC-8 checksum, followed by an optional payload and an optional checksum for the payload. Packet headers all start with a single byte indicating the type of packet. Message packets can be prepended with meta-data packets such as destination and source packets. Packet header types, with their corresponding hexadecimal values, are shown in table 4.1.

Table 4.1: Header types

Header	Hexadecimal Value
ACK	`0x00`
SEGACK	`0x80`
PACKET	`0x01`
SEGMENT	`0x02`
SEGMENT (Last)	`0x42`
SEGMENT (With Acknowledgement)	`0x82`
SOURCE	`0x03`
DEST	`0x04`
EXPIRE	`0x05`

The acknowledgement packet (figure 4.4(a)) is sent after successful reception of a message packet. Segmented acknowledgement packets (figure 4.4(b)) are used only when progress is requested with the send of a segmented packet.



(a) Standard acknowledgement (b) Segmented acknowledgement

Figure 4.4: Acknowledgement packets

Message packets, when not segmented, have a limit of 64Kb for a payload. Figure 4.5 shows the standard unsegmented message packet definition. Segmented packets have three types namely the first segment, intermediary segments and the last segment. During standard operation an acknowledgement is sent only for the last segment, unless the bit SEG is set in the segment header. Last segments set the LAST bit in the packet header to indicate the end of the stream.

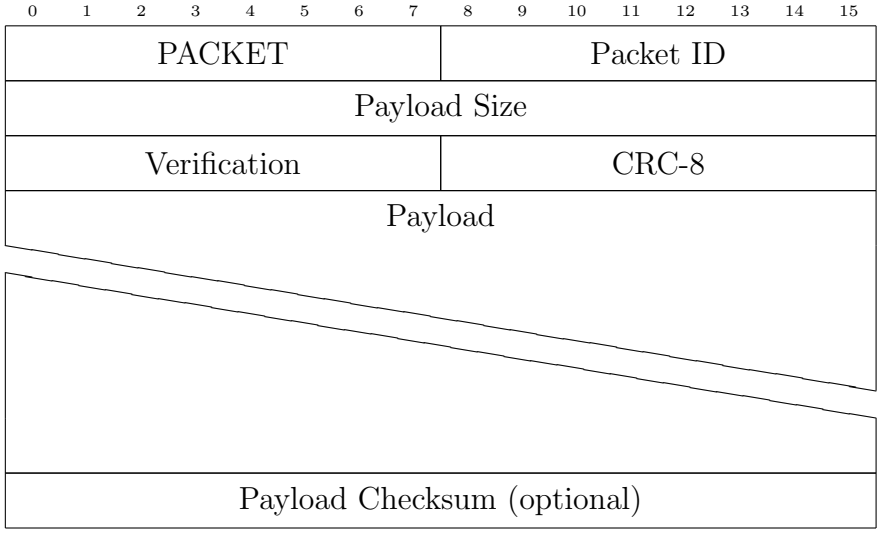


Figure 4.5: Standard unsegmented packet

First segments can be sent to the server with any segment size and without initially specifying the total payload size, as a last segment will signify the end of the data stream. In the case where an endpoint chooses to omit the total size, the field must be set to 0. Segmented packets' definition is shown in figure 4.6.

In order to forward packets to a recipient or recipients, a special forward packet is defined in figure 4.7. Multiple forward packets can be chained to forward a packet to multiple destinations. A single forward packet may also contain a group name to forward to a whole group. The packet header type allows for a special bit (CRS) to be set in the case of cross account forwarding. Forward packets contain a special field for additional forwarding meta-data: a null-terminated string that can be used to chain an entire routing table for internal forwarding. Source packets (figure 4.8) show the source endpoint that sent the packet. Source packets follow the same principle as destination packets.

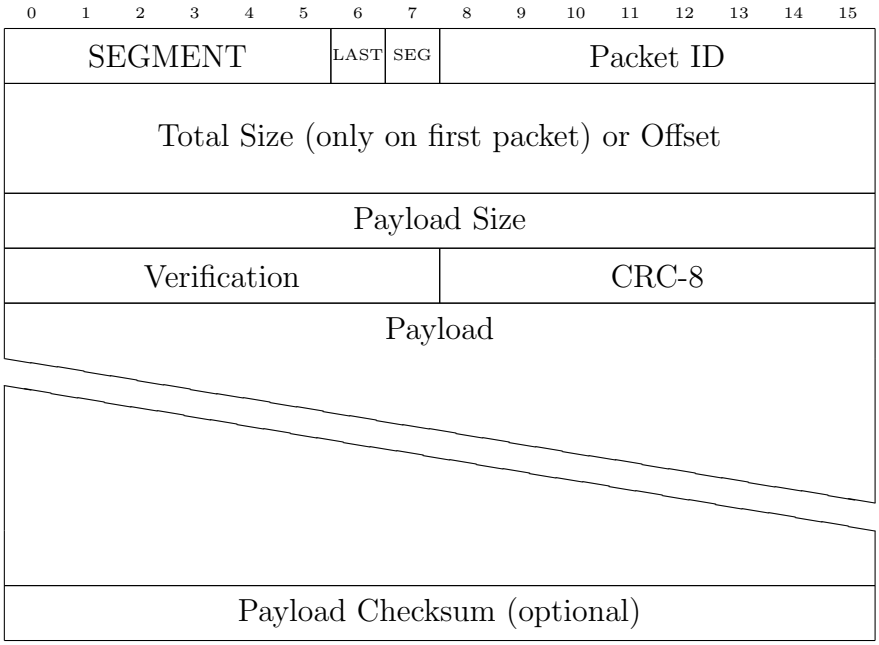


Figure 4.6: Segmented packet

Message packets can be set to expire when unsent for a specific time since the server has received the packet. These expiry packets have the definition shown in figure 4.9. The expire time is given with 5-minute granularity allowing for 227 days as the maximum expiry time.

All message packets receive a 1-byte session unique incremental ID. There can be at most 255 active unacknowledged packets. If more packets are sent, then an ambiguity occurs, upon which the endpoint disconnects and requests the last ID, as it could refer to the initial packet or the last packet. Acknowledgements are sent in chronological order to improve ease use.

4.2.1.3 User Defined Packets

With the data management system protocol, as defined above, it is straightforward to encapsulate user defined packets within the payload of message packets. This provides any user of the protocol with a system with guaranteed delivery of packets and forwarding capabilities.

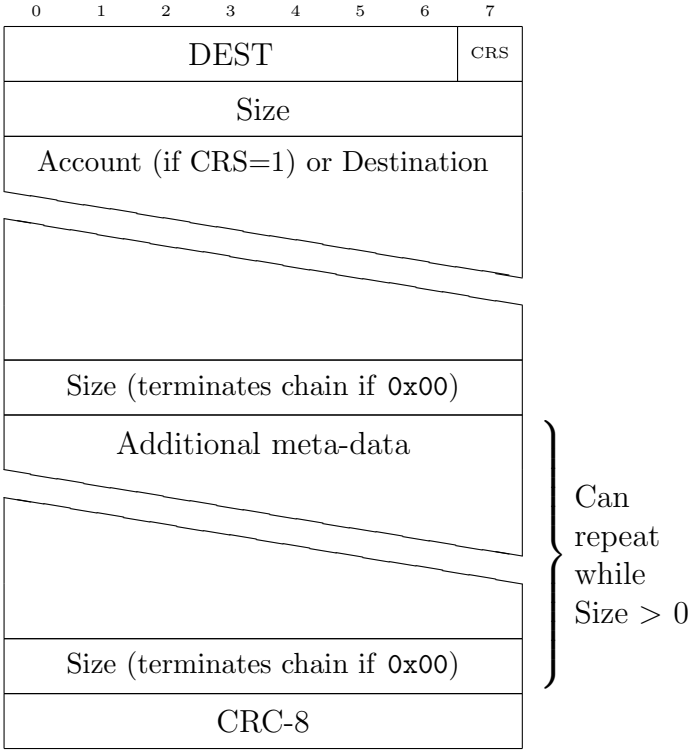


Figure 4.7: Destination packet

4.2.2 Storage System

As MongoDB was selected as the database management system in Chapter 2, the functional units below describe MongoDB's integration into the system architecture.

4.2.2.1 Query Router (F/U 2.1)

For larger deployments where shards are available, the `mongos` router application from MongoDB can be used. It is a query router created specifically for sharded MongoDB deployments. The C++ MongoDB driver can connect to `mongod` and `mongos` processes.

4.2.2.2 Storage Nodes (F/U 2.2)

Storage nodes are `mongod` processes in a sharded and/or replicated setup depending on the choice of deployment.

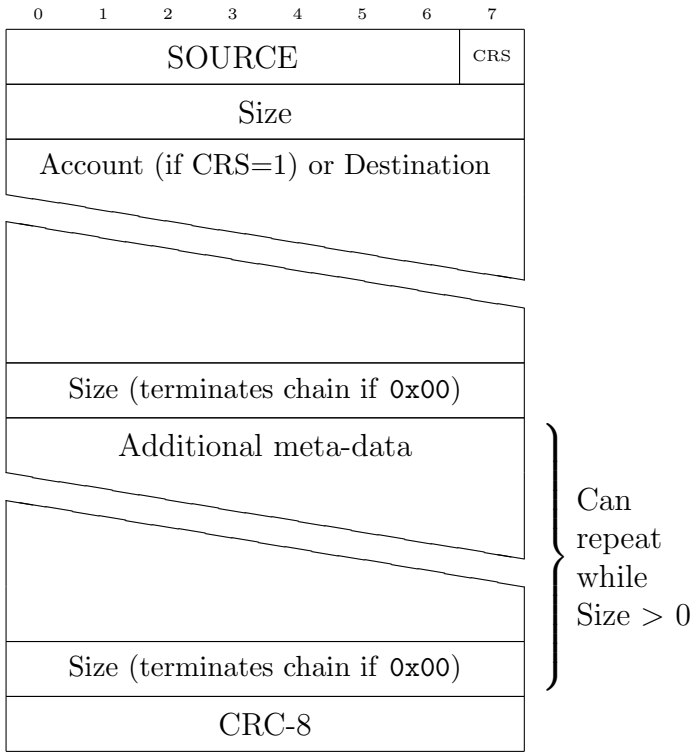


Figure 4.8: Source packet

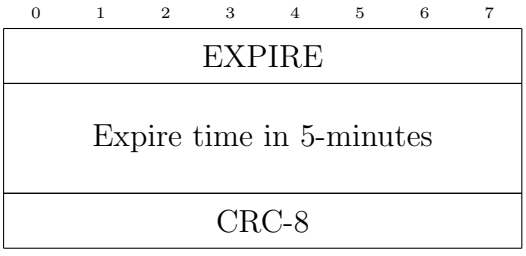


Figure 4.9: Expiry packet

4.2.2.3 Data Replicator (F/U 2.3)

MongoDB replica sets provide the ability to replicate data among themselves in a master-slave configuration. Each query can be specified to read from a replica to leverage the redundancy for additional read throughput or from a master for increased consistency. Writes and updates can specify the consistency level that defines how many (if any) replicas must have the data before indicating a successful operation.

4.2.2.4 Diagnostics (F/U 2.4)

Profiling data can be enabled on each `mongod` process that logs operations that take longer than a specified time[1]. Furthermore MongoDB provides `mongostat` and `mongotop` that are applications which show usage statistics of MongoDB.

4.2.3 Data Management System

C++ is used as the programming language for the system and uses publicly available libraries to extend C++'s standard libraries. Libraries used include OpenSSL for cryptographic functions, Boost for asynchronous network sockets and threads, and MongoDB's C++ client driver.

4.2.3.1 Connection Management (F/U 1.1)

Socket operations are mostly blocking in nature but Boost provides an event-driven asynchronous socket system named ASIO[2] (Asynchronous Input/Output). Event-driven software often have a lot of idle time and it is for this reason that a pool of threads is used in the system rather than a thread per connection. During high load, more threads can dynamically be added to the pool to service more endpoints, but without more physical processors no performance increase will be achieved. MongoDB's client driver operations, however, are blocking and thus more threads will be beneficial in this scenario, regardless of the number of physical processors.

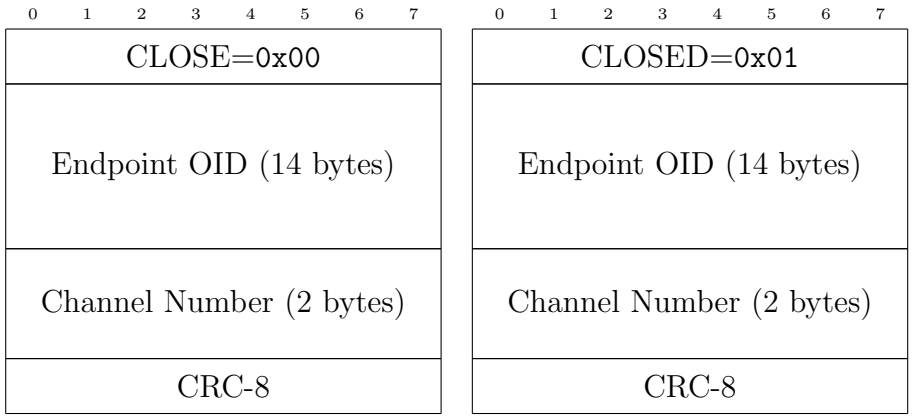
Embedded systems are often mobile and thus suffer from undetectable connection loss that can cause invalid idle connections. As mentioned in the preliminary synthesis all connections are monitored for idle time. If the idle time exceeds 30 minutes, the connection is closed by the server.

[1] Default 10ms

[2] `http://www.boost.org/doc/libs/1_49_0/doc/html/boost_asio.html`

As each endpoint can have multiple channels, each connection is stored in a hash-table with the hash-key created on the endpoint's OID and channel number. This allows for fast access to a specific connection object for management purposes. Each connection's details are also logged to the database for auditing purposes.

During the authentication phase the database is checked for consistency. If it is detected that the connection has previously been owned by another node, the node is contacted with a UDP packet to request that the connection should be closed. When the connection is closed (or if it is already closed) on the previous owning node, the node will respond with another UDP packet indicating that the connection is closed. The UDP packet definitions are shown in figure 4.10.



(a) Request connection close packet definition.

(b) Acknowledge close of connection packet definition.

Figure 4.10: Connection closing packets.

The connection manager periodically polls the storage system for any new packets in each connection's outgoing queue. In order to improve the latency of the system, nodes send a UDP packet (figure 4.11) to the node who owns the connection of an endpoint that should receive a new packet. This makes the packet forwarding system event-driven with a polling fallback. For this reason the polling period can be set to 30 seconds.

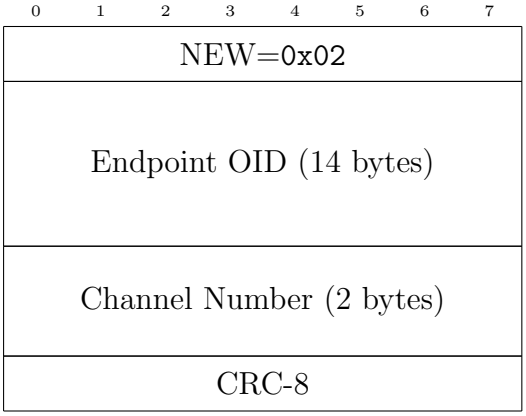


Figure 4.11: New data arrived packet definition.

4.2.3.2 Protocol Handler (F/U 1.2)

In addition to implementing the protocol, the protocol handler also enforces a rate limit according to each account's QoS agreement. The rate limit ensures that endpoints cannot transmit or receive more data than the QoS agreement stipulates.

F/U 1.2 also makes use of the database interface (I/F 2) to write status changes of packets to ensure a consistent state for each endpoint/channel. It is imperative to have a consistent state if an endpoint connects to a different node, as an ambiguity can lead to data loss.

The protocol handler concurrently handles received and transmitted packets. Received packets follow the states in figure 4.12 and transmitted packets follow the states in figure 4.13. A typical message sequence is shown in figure 4.14 where two packets are exchanged between an endpoint and the server.

4.2.3.3 Node Management (F/U 1.3)

Most of the node interoperation functions have already been discussed in F/U 1.1. One aspect still remaining is the node heartbeat system. For the interoperability function to work consistently, it is critical to be able to determine the health status of another node. Each node will be assigned a unique ID. With the storage system as the central point, all nodes can periodically update their health status on the server. If an index on the heartbeat timestamp and node ID is created, it is relatively easy to determine the health status of any other node.

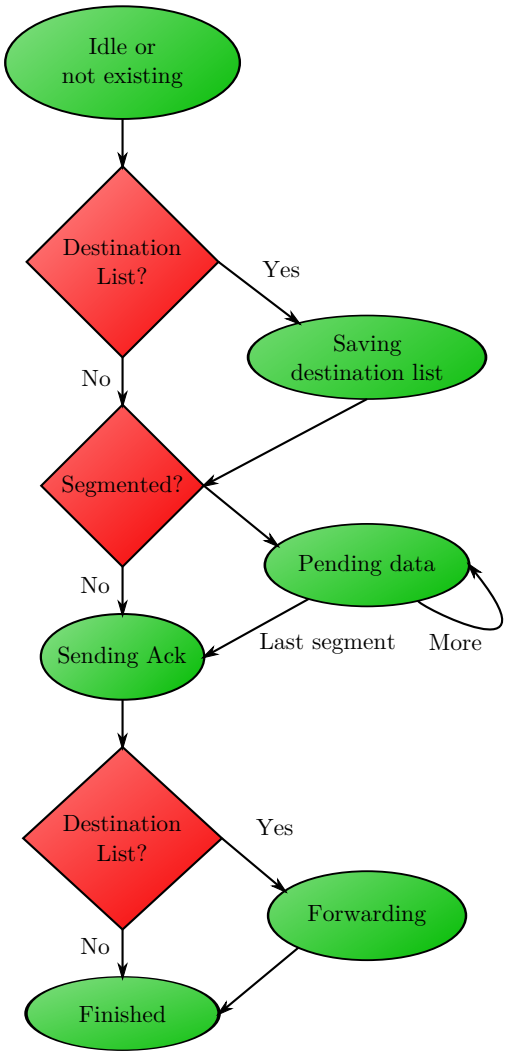


Figure 4.12: Received packet states

If a node's last heartbeat timestamp is older than a specific duration, the node can be seen as offline. This status is essential to determine when a connection is started on a new node. After a node has sent a CLOSE request to the original owner of a connection, the node starts a timer. If the timer expires before receiving the CLOSED response, the node will check the health status of the previous owner. If the health check fails, the old node is assumed inactive and the connection can be resumed. If the health check passes, a new CLOSE message is sent again.

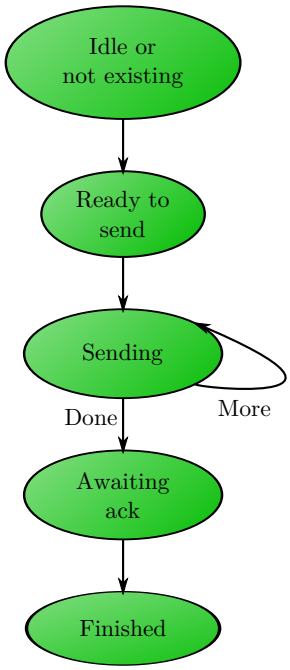


Figure 4.13: Transmitted packet states

4.2.3.4 Billing (F/U 1.4)

Billing is based on the QoS arrangement in the SLA. This implies that data is billed per quantity depending on the agreed rate-limit. Also, every forwarded packet is billed a small fixed amount.

In order to aggregate all the billing data, it is required that the database keeps record of every sent, received and forwarded packet. MongoDB's map/reduce utility provides the aggregation capability required. The map/reduce and finalize functions are shown in listing 4.1. Two iterations, that are reduced together, are required to aggregate both transmitted and received data. The functions `maprx` and `maptx` emit received and transmitted data respectively. All emitted data is reduced by `reduce`.

This map/reduce job can be run on a daily basis, each day accumulating the data count. The query field of the map/reduce function is also incremented daily to only measure new data. At the end of the month the accumulated data can be used in the billing report sent to the account holder.

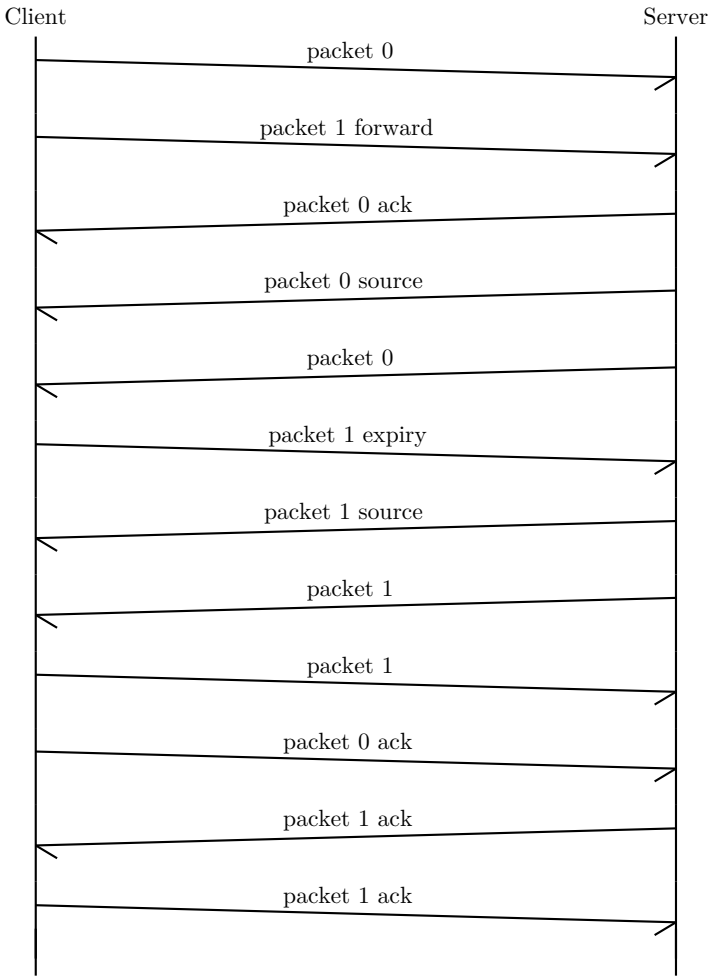


Figure 4.14: Protocol message sequence example

4.2.3.5 Diagnostics (F/U 1.5)

A special logging library is used[3] to improve the logging capabilities of the system. The library allows for multiple streams to be written to with a single log, which implies that file, network and console logging can be done without modifying any code. A verbosity level can be specified to allow for flexible output options. By utilizing the logging library, diagnostic data can be logged to a file that can be monitored periodically for any errors or irregular performance issues.

[3] `http://sourceforge.net/projects/cppmylogger/`

```
function maprx()
{
    emit(this.srcaccount,
         {rxdata:this.total, rxcount:1, forwards: this.dests});
}

function maptx()
{
    emit(this.srcaccount,
         {txdata:this.total, txcount:1});
}

function reduce(key, vals)
{
    var r = {rxdata:0, rxcount:0, forwards: 0, txdata: 0, txcount: 0};
    values.forEach(function(v)
        {
            r.rxdata += v.rxdata;
            r.txdata += v.txdata;
            r.forwards += v.forwards;
            r.rxcount += v.rxcount;
            r.txcount += v.txcount;
        });
    return r;
}
```

Listing 4.1: Billing map/reduce functions

4.2.4 Database Interface

As MongoDB was chosen as the DBMS, all database communication utilizes the MongoDB C++ client driver. In sharded deployments the `mongos` router application is used as the connection endpoint, otherwise a direct connection to the `mongod` server application will be made.

The database used by the system contains the following collections, with the same basic structure as defined in the preliminary synthesis:

- Accounts;
- Endpoints;
- Connections;
- RXPackets;
- TXPackets;

- Billing;
- Nodes.

4.3 Deployment

The deployment options are endless depending on the requirements of the system. For small deployments it is entirely feasible to have a single node with a single storage system. For larger deployments it is required to have a load balancer before the data management systems in order to distribute the load. Large systems also require multiple storage nodes in a sharded and replicated fashion to scale on par with greater write and read requirements.

For the purpose of this research project, an illustration of a large deployment on Amazon EC2 is shown in figure 4.15. The front-end to the system is a set of elastic load balancers (ELB's) that are all listed in the Route 53 DNS records for the deployment's domain name. This implies that requests will be given the IP-address of an ELB in a round-robin fashion. The DNS is the first tier of load balancing followed by the ELB's as the second tier. Each load balancer has a number of data management system nodes to distribute its load to in a round-robin fashion.

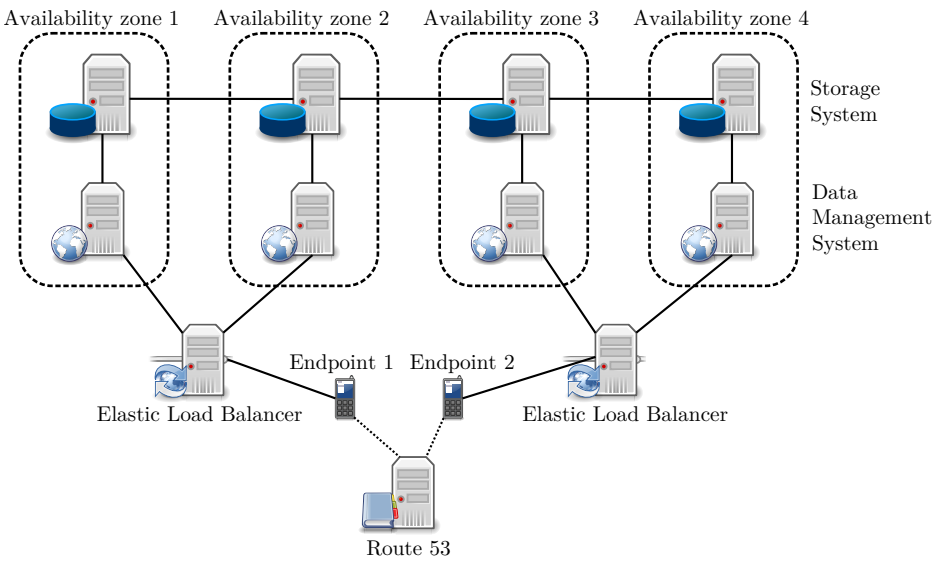


Figure 4.15: AWS large deployment example

In order to make the entire system scalable, the storage system consists of multiple shard nodes, with each shard having two replicas for improved redundancy and read-throughput.

Additionally, CloudWatch is used to monitor the load of the system. If the storage nodes' disk operations or CPU-usage exceed a threshold value, more storage nodes (shards) are started to distribute the data load. Likewise, if any data management system node exceeds a threshold value, more nodes will be started and linked to an ELB.

The system is thus automatically infinitely scalable and highly available.

4.4 Evaluation

The evaluation of the data management system is performed in this section. This is done by ensuring all requirements have been addressed functionally and performance-wise.

4.4.1 Functional Capability

4.4.1.1 Authentication

The handshake module allows for authentication of both the endpoint and of the server.

4.4.1.2 Session Support

By maintaining a persistent and consistent state in the storage system, the handshake phase of the protocol provides session support.

4.4.1.3 Guaranteed Delivery of Packets

The protocol utilizes TCP's per session guaranteed delivery of packets along with the persistent state updated in the storage system to guarantee delivery of packets.

4.4.1.4 Segmented Packet Support

The protocol and storage system provide support for segmented packets.

4.4.1.5 Persistent Storage of Data

Persistence of data is achieved by the storage system as MongoDB is a persistent data store.

4.4.1.6 Packet Forwarding

Both the protocol and the storage system allow for forwarding of packets. Nodes can interact to improve the latency between receiving a packet and forwarding the packet.

4.4.1.7 QoS Based Data Billing

The billing module makes use of the stipulated rate-limit to achieve QoS based billing of data usage. MongoDB's map/reduce functionality makes the process of aggregating the billed data effortless.

4.4.2 Performance Characteristics

4.4.2.1 Ease of Use

The protocol is easy to implement in any embedded system that supports networking. SHA-256 is the only CPU-intensive operation, but only occurs twice per handshake[4].

Simplistic header-body-footer design of the protocol packets and the use of an incremental ID per message packet further improve the usability.

4.4.2.2 Availability

Deployment of a highly available system is achieved by utilizing cloud technology. The shared-nothing architecture, that this system inherently has, makes fail-over support trivial.

4.4.2.3 Scalability

By leveraging cloud services and architectures the system is infinitely scalable. By adding more nodes to load balancers, a greater load can be distributed. Adding more load balancers also increases the total capability of the system. In order to scale on par with the data management system, the storage system can add more shards and replicas.

[4]The endpoint's hash and the server's expected hash.

4.4.2.4 Security

The inherent security provided by cloud services, such as AWS, allows applications to run securely. Furthermore the handshake phase of the protocol securely authenticates both ends of communication. To increase security the entire protocol can be encapsulated in TLS/SSL for encryption of data, due to it already being TCP/IP based.

4.5 Summary

Chapter 3 defined a preliminary architecture that served as an input to the detail synthesis phase, with the preliminary architecture as basis. Each element in that preliminary architecture was described in more detail in this chapter.

By evaluating the detail synthesis with respect to the defined research requirements, it was shown that the synthesis is valid. Table 4.2 shows requirements addressed by each function or interface.

With the synthesized system complete and evaluated, tests were performed to empirically verify each functional capability and performance, and to collectively validate the data management system.

Table 4.2: Requirements vs. architecture elements matrix

	F/U									I/F	
Requirement	1.1	1.2	1.3	1.4	1.5	2.1	2.2	2.3	2.4	1	2
Authentication	x	x	-	-	-	-	-	-	-	x	x
Session support	x	x	x	-	-	-	-	-	-	x	x
Guaranteed delivery	-	x	-	-	-	-	-	-	-	-	x
Segmented packets	-	x	-	-	-	-	-	-	-	x	x
Persistent storage	-	x	-	-	-	x	x	x	-	-	x
Packet Forwarding	-	x	x	-	-	-	-	-	-	x	x
QoS billing	x	x	-	x	-	-	-	-	-	x	x
Ease of use	-	-	-	-	-	-	-	-	-	x	-
Availability	x	-	x	-	x	x	x	x	x	-	x
Scalability	x	-	x	-	x	x	x	x	x	-	x
Security	x	-	-	-	x	-	-	-	x	x	x

I didn't fail the test, I just found 100 ways to do it wrong.

Benjamin Franklin

Chapter 5

Empirical Tests and Results

5.1 Overview

With chapters 3 & 4 having defined the system that addresses the research problem, practical tests were performed. *The process of verification and validation for this research project that was followed, is presented in this chapter by evaluating practical test results with respect to the research requirements.*

All tests and experiments are fully described and results are provided, followed by a summary. Functional and performance tests individually verify the synthesized system, while the system is validated from collective functional capability and performance measurements. *Together with the functional and performance test results, field test results finally validate the data management system.*

5.2 Tests

Due to the nature of client/server applications, it is not trivial to perform unit tests. Functional capability tests have to be performed using both the server, client and a consistent database setup concurrently. Performance characteristic tests additionally have to be controlled in order to accurately measure the metrics, and may require multiple client applications.

By using debugging breakpoints at critical points in the server and client code, and manipulating the database, certain scenarios can be simulated for robustness tests.

To aid the testing process, a client-side API was developed in Microsoft .NET C#. The API implements the client-side protocol and relevant network functions. A client application was developed with the API in order to run the tests against the system.

A number of empirical tests are defined below, to provide verification evidence of the complete system in terms of functional capability and performance characteristics. For the sake of brevity, the following common features of the tests are defined here:

- Timing measurements at key points were measured and logged;
- Breakpoints at key points were used to check if every process occurs safely and consistently for robustness;
- All key events were logged;
- Database sizes were logged.

5.2.1 Functional Capability

Table 5.1 shows which functions were tested by each functional capability test defined in the tests below. These tests' results were used to verify the data management system's functional capability.

Table 5.1: Test/functional capability matrix

Function	Functional Test 1	2	3	4	5	6	7
Authentication	x	x	x	x	-	-	-
Back-off	x	-	-	-	-	-	-
Clean session	x	x	x	x	-	-	-
Continue session	-	x	-	-	-	-	-
Packet send	-	-	x	x	-	-	-
Packet receive	-	-	x	x	-	-	-
Segmented packet send	-	-	x	x	-	-	-
Segmented packet receive	-	-	x	x	-	-	-
Packet acknowledgement	-	x	x	x	-	-	-
Packet forwarding	-	-	-	x	-	-	-
Packet group forwarding	-	-	-	x	-	-	-
Packet cross-account forwarding	-	-	-	x	-	-	-
Forwarded packet expiry	-	-	-	x	-	-	-
Rate-limiting	x	x	x	x	-	-	-
Server heartbeat	-	-	-	-	x	-	-
Account management	-	-	-	-	-	x	-
Endpoint management	-	-	-	-	-	x	-

continued on next page...

Table 5.1: Test/functional capability matrix (continued)

Function	Test 1	2	3	4	5	6	7
Channel management	-	-	-	-	-	x	-
Group management	-	-	-	-	-	x	-
Billing generation	-	-	-	-	-	-	x

Test 1

Aim

To test the handshake module in both the server and client. The handshake module is responsible for establishing authentication and exchanging session information which were both requirements for the system. The following functions were verified by this test:

- Authentication;
- Back-off;
- Clean session;
- Rate-limiting.

Setup

The test was run four times, each with a pristine database set up differently to test each possible scenario for the handshake protocol.

a Endpoint set up with only an account wide password.

b Endpoint set up with its own password.

c Endpoint set up with an incorrect password.

d Endpoint set up with a back-off time.

Methodology

This test followed the same methodology for all cases. The client connected to the server with the database set up for each case and the results were logged.

Expected Outcomes

a,b The connection should be successfully established and in an idle state. The client's detail for the connection should be logged to the database.

c The server should terminate the client's connection and log the occurrence to the database. The client should automatically try to reconnect.

d The server should wait for the client to terminate. The client should terminate immediately and reattempt connection automatically after the back-off period expires. Upon reconnection before that time (that can be forced in the client application), the server should terminate the connection.

Test 2

Aim

To test the session negotiation module in both the server and client. This includes cleaning and restoring sessions which were both needed for the session support requirement. The following functions were verified by this test:

- Authentication;
- Clean session;
- Continue session;
- Packet acknowledgement;
- Rate-limiting.

Setup

A pre-generated database was used that already defined incoming and outgoing packets for endpoints in all possible scenarios. The scenarios are listed here:

- No incoming or outgoing packets;
- Only incoming packets;
- Only outgoing packets;
- Incoming and outgoing packets;
- Incoming and outgoing packets with partial progress.

Each scenario was tested during a set of cases, as defined here:

a Client initiated a clean session.

b Server initiated a clean session.

c Client requested to resume a valid session.

d Client requested an invalid session to resume.

Methodology

Each test case was run against the same database, that means after each case the database was restored. The client's endpoint was adjusted according to the cases and scenarios mentioned before. This implies that the test was run twenty times.

Expected Outcomes

a The queues of both the server and endpoint should be consistent. The database should reflect that all pending packets (if there were any in the scenario) have been flagged as cancelled.

b The server should wait for the client to disconnect. The client should, however, disconnect immediately and reattempt the connection. If the client does not disconnect within one minute, the server terminates the connection regardless. The same outcomes of **a** should then be reached after reconnection.

c The queues of both the server and endpoint should be consistent. The database should reflect that complete packets were acknowledged and pending packets were resumed.

d The server should detect inconsistencies and force a clean session that should have the same outcomes as **b**.

Test 3

Aim

To test the packet send and receive modules. This also includes sending and receiving of acknowledgements. Packet and segmented packets were requirements for the system. The following functions were verified by this test:

- Authentication;
- Clean session;
- Packet send;
- Packet receive;
- Segmented packet send;
- Segmented packet receive;
- Packet acknowledgement;
- Rate-limiting

Setup

Four iterations of the test were run. Every test recreated the database into a pristine condition. The scenarios for each iteration are given below:

a Server sends 1000 packets to a single endpoint.

b Endpoint sends 1000 packets to the server.

c Server sends 1000 segmented packets to a single endpoint.

d Endpoint sends 1000 segmented packets the server.

Methodology

A pristine database was loaded for each scenario, which defined a single endpoint with an empty packet queue. Each iteration verified the integrity of the packet by using the appended CRC-8 checksum. Acknowledgements were sent after successfully persisting the data to the database when the server was the recipient.

Expected Outcomes

The outcomes should be the same for all scenarios:

- Database must be consistent;
- Message IDs must wrap correctly;
- Invalid IDs must terminate the connection (from either end);
- Acknowledgements must reflect the correct IDs of packets;
- Packet integrity checks that fail (that is forced during test) must terminate the connection (from either end).

Test 4

Aim

To test packet forwarding. Packet forwarding was a requirement for the system. The following functions were verified by this test:

- Authentication;
- Clean session;
- Packet send;
- Packet receive;
- Segmented packet send;
- Segmented packet receive;
- Packet acknowledgement;
- Packet forwarding;
- Packet group forwarding;
- Packet cross-account forwarding;
- Forwarded packet expiry;
- Rate-limiting.

Setup

This test considered seven specific scenarios in which the test had to succeed:

a Packet to a single destination.

b Packet to a single group destination.

c Segmented packet to a single destination.

d Segmented packet to a single group destination.

e Packet to multiple destinations.

f Segmented packet to multiple destinations.

g Packets with varying expiry times.

A single endpoint always serves as a packet source. Depending on the scenario, multiple clients concurrently serve as recipients of packets.

Methodology

A database, that contained groups and multiple endpoints, was loaded. Each scenario was then tested without resetting the database. All clients were executed concurrently with a single server application. For scenario **g** the destination endpoint should not have executed until the predetermined time interval had expired.

Expected Outcomes

The outcomes should be the same for all scenarios:

- Database must be consistent;
- Messages delivered correctly;
- Destination chains were appended correctly;
- Source endpoint meta-data was correctly forwarded.

Scenario **g** should not have delivered the packet and the database should have reflected that the packet had expired.

Test 5

Aim

To test server heartbeat capability. Server heartbeat capability is essential for scalability as a requirement of the system, and was verified in this test.

Rationale

Heartbeat capability is essential for utilizing the UDP performance enhancements. Furthermore, heartbeat capability is critical for endpoint hand-off.

Setup

This test required multiple server applications and a pristine database. Each server had a unique identifier and IP-address/port combination.

Methodology

The servers were run concurrently and after all servers had become aware of every other server, a random group of servers was closed.

Expected Outcomes

All the servers should have become aware of every other server within $2 \cdot t_{heartbeat}$ seconds. Upon closing the group of servers, the remaining servers should have become aware of the closed servers within $2 \cdot t_{heartbeat}$ seconds.

Test 6

Aim

To test the management functions of the system. Although this was not a requirement for the research, it was needed for the system to be practically feasible. The following functions were verified by this test:

- Account management;
- Endpoint management;
- Channel management;
- Group management.

Setup

This test ran a single iteration, where the database had to be empty.

Methodology

The following tasks on the front-end were tested by means of unit testing the CRUD (Create, Remove, Update, Delete) operations:

- Account management;
- Endpoint management;
- Endpoint channel management;
- Group management.

Expected Outcomes

The database had to reflect all the CRUD operations and document contents in the database had to be consistent with the relevant models.

Test 7

Aim

To test the billing function of the system. Billing was a required function of the system and was verified in this test. QoS was handled during performance tests by implementing rate-limiting and was not tested here.

Setup

This test ran a single iteration where the database contained the following:

- Sent and received packet data in the same account;
- Cross-account sent and received data;
- Map-reduce results (bogus) of a previous month's billed usage.

Methodology

As billing data is aggregated in MongoDB, this test only required a map-reduce operation. The resulting map-reduce collection contained the newly accumulated and monthly compounded usage reports.

Expected Outcomes

The result of the map-reduce operation should have contained the accumulated monthly compounded usage reports.

5.2.2 Performance Tests

Performance tests require a controlled environment. Any uncontrolled or unmonitored variables can lead to invalid and conflicting results. The tests were therefore run on a single machine installation, where parameters of the environment could be controlled, manipulated and measured. Network latency also had no effect on a single machine.

Latency, throughput and concurrency are three metrics that are dependent on each other, therefore a single test was defined that tested all three metrics simultaneously.

Performance Test: Latency, Throughput and Concurrency

Aim

Latency, throughput and concurrency impact the scalability of the system. Scalability was a requirement for this system.

The latency here refers to the round-trip time (RTT) for a packet from an endpoint to the server and then for the acknowledgement to propagate back to the endpoint. This gives an indication of the processing time for a single packet. The latency also refers to the RTT for server to endpoint packets. The delay when forwarding a packet was determined empirically. Network latency was not taken into consideration as it is deployment specific and not an indication of the system's intrinsic performance.

The throughput metric of the system defines both the maximum and average transfer rates in terms of the data rate (KB/s) and packet rate (packets/s). This metric was derived from the latency metric.

Concurrency defines how many endpoints can be serviced simultaneously by the system and is dependent on the throughput and latency relative to the number of endpoints. Concurrency will thus be a function of the desired throughput.

Latency, throughput and concurrency of the system were verified by this test.

Setup

Multiple endpoint applications on the same physical machine as the server, were required. The test ran multiple iterations to acquire a more accurate sample, with each iteration having varied the number of concurrent endpoints and packet size.

Methodology

The test was run in multiple iterations, one for each variable combination. Each iteration was run in two phases, each with a total of 512 packets. Firstly, only the endpoint-to-server packet tests were run, followed by server-to-endpoint packets. A separate packet generator application was used to inject the packets into the database's outgoing queue for the endpoints during the server-to-endpoint phase. By testing the metric in two phases, empiric RTT metric for packets in any direction could be determined.

Timestamps were logged programmatically for each iteration at the following key-points:

- Endpoint 1 sent the packet (t_{tx});
- Server received the data (t_{rx});
- Server sent the acknowledgement (t_{txack});
- Endpoint 1 received the acknowledgement (t_{rxack});
- Server sent the forwarded packet to endpoint 2 ($t_{tx'}$);
- Endpoint 2 received the data ($t_{rx'}$);
- Endpoint 2 sent the acknowledgement ($t_{txack'}$);
- Server received the acknowledgement ($t_{rxack'}$).

Outcomes

With the timestamps defined above, the following latencies could be calculated:

- Latency for endpoint to server packet:

$$t_{rxack} - t_{tx}$$

- Packet processing and database storage time:

$$t_{txack} - t_{rx}$$

- Network latency for endpoint to server packet:

$$(t_{rxack} - t_{txack}) + (t_{rx} - t_{tx});$$

- Latency for server to endpoint packet:

$$t_{rxack'} - t_{tx'}$$

- Network latency for server to endpoint packet:

$$(t_{rxack'} - t_{txack'}) + (t_{rx'} - t_{tx'})$$

- Total round-trip-time for an endpoint-to-server packet:

$$t_{rxack} - t_{tx}$$

- Total round-trip-time for a server-to-endpoint packet:

$$t_{rxack'} - t_{tx'}$$

It should be noted that the network latency mentioned here will give an indication of the overhead of the socket processing and is not a metric of the physical network interface. It is assumed that $t_{rx'} = t_{txack'}$. Endpoint 1 and 2 can be the same endpoint.

Forwarded packet routing latency in this test should have been negligible and would generally be equal to the propagation time of the UDP notification packet between servers. If UDP notification failed, the latency would have been equal to the polling fallback time of the server.

5.2.3 Database Tests

Since MongoDB is used as the DBMS, the test suites[1] defined by MongoDB were used to test the database. The test suites are listed here:

- C++ unit tests;
- Core tests;
- Replication tests;
- Sharding tests.

Only the production version (v2.2.0) of MongoDB is used, which implies all test suites succeeded at the time of release.

5.2.4 Availability

Availability was tested in functional capability test 5. By design, the availability of the system is dependent on the heartbeat period, the number of redundant systems and the physical deployment scheme. When a server becomes unavailable another system should take over within a period of $2 \cdot t_{heartbeat}$ seconds.

5.2.5 Scalability

Scalability is difficult to measure as it is dependent on the deployment. On Amazon's EC2, scalability can be configured and fixed. The system is scalable by design due to its core components having a shared-nothing architecture.

[1] `http://www.mongodb.org/display/DOCS/Smoke+Tests`

5.3 Results

5.3.1 Functional Capability

All functional capability tests passed. This verifies that the data management system's functional implementation is correct. Table 5.2 shows the summarized results.

Table 5.2: Test/functional capability results matrix

	Functional Test						
Function	1	2	3	4	5	6	7
Authentication	✓	✓	✓	✓	-	-	-
Back-off	✓	-	-	-	-	-	-
Clean session	✓	✓	✓	✓	-	-	-
Continue session	-	✓	-	-	-	-	-
Packet send	-	-	✓	✓	-	-	-
Packet receive	-	-	✓	✓	-	-	-
Segmented packet send	-	-	✓	✓	-	-	-
Segmented packet receive	-	-	✓	✓	-	-	-
Packet acknowledgement	-	✓	✓	✓	-	-	-
Packet forwarding	-	-	-	✓	-	-	-
Packet group forwarding	-	-	-	✓	-	-	-
Packet cross-account forwarding	-	-	-	✓	-	-	-
Forwarded packet expiry	-	-	-	✓	-	-	-
Rate-limiting	✓	✓	✓	✓	-	-	-
Server heartbeat	-	-	-	-	✓	-	-
Account management	-	-	-	-	-	✓	-
Endpoint management	-	-	-	-	-	✓	-
Channel management	-	-	-	-	-	✓	-
Group management	-	-	-	-	-	✓	-
Billing generation	-	-	-	-	-	-	✓

5.3.2 Performance Characteristics

The performance characteristics results are discussed in the sections below. Latency, throughput and concurrency test metrics are illustrated separately.

5.3.2.1 Latency

Figure 5.1 shows the effective per packet latency for different packet sizes and number of concurrent endpoints. A linear relationship between effective "per packet" latency and packet size exists irrespective of the number of endpoints. It can be seen from the results that the latency slope decays exponentially with more endpoints, which implies that packet sizes has less influence on performance than the number of endpoints. The results also show that server-to-endpoint packets have a much smaller latency.

5.3.2.2 Throughput

Figure 5.2 shows the total throughput of the system. Examining the results, reveals that with an increase in packet size, the data throughput follows a logarithmic-type growth curve. It can also be noted that more endpoints deliver a greater throughput, which is the expected behaviour. The peak throughput will, however, reach a maximum value, which is due to the physical limitations of the system.

Packet throughput results are shown in figure 5.3. The throughput per packet decays exponentially with an increase in packet size. This behaviour is expected due to the increased network traffic and hard-drive latencies. It is, however, still evident that an increased throughput is reached with more endpoints.

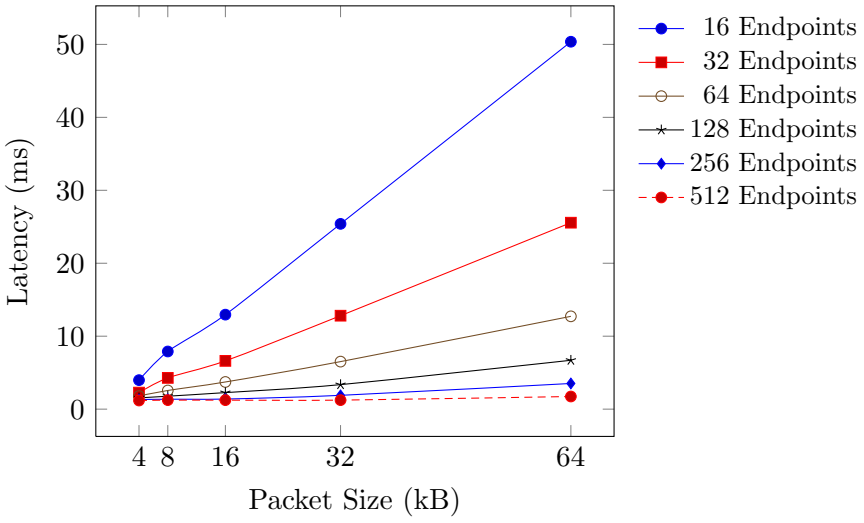
5.3.2.3 Concurrency

In order to illustrate the concurrency of the system, the results of the data throughput tests are shown in figure 5.4, but with the values scaled relative to 512 endpoints. The results show the relative decline in throughput as more endpoints use the system.

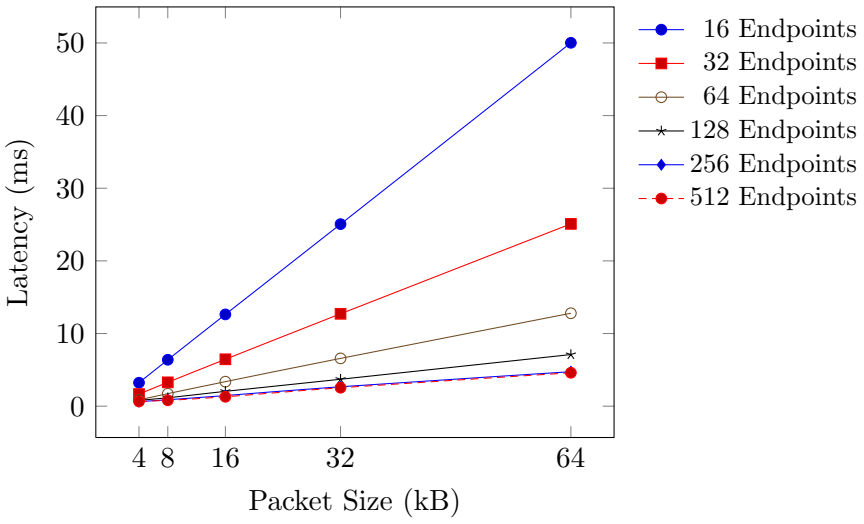
From the results at 64kB incoming packets, it can be seen that the relative efficiency of 16 units to 512 units is only 2.942 even though 512 endpoints are 32 times more units. This result shows that the system scales well under increased load, which implies good concurrency and therefore scalability.

5.3.2.4 Availability & Scalability

From the shared-nothing architecture of the system and the cloud-based deployment, the system is inherently available and scalable. All components are capable of scaling on demand and redundant nodes can be used for higher availability. Even with a single node system the performance scaled well with an increase of endpoints.

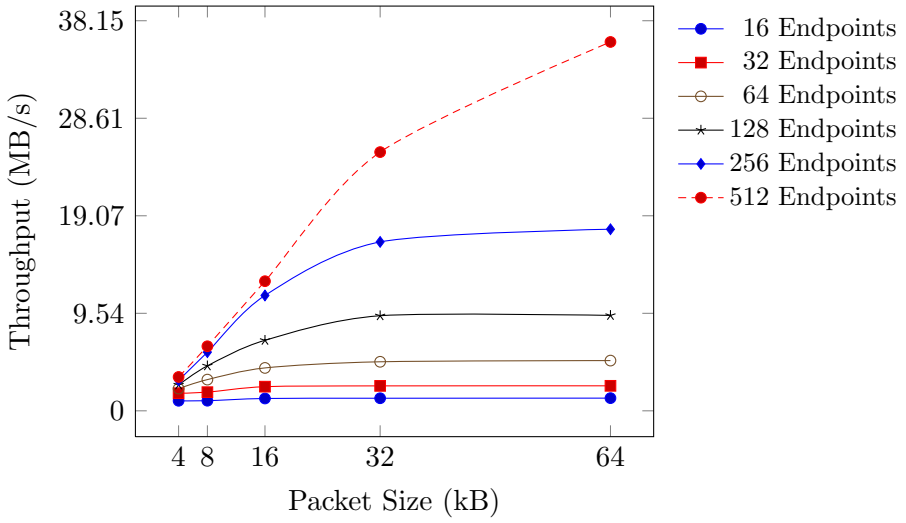


(a)

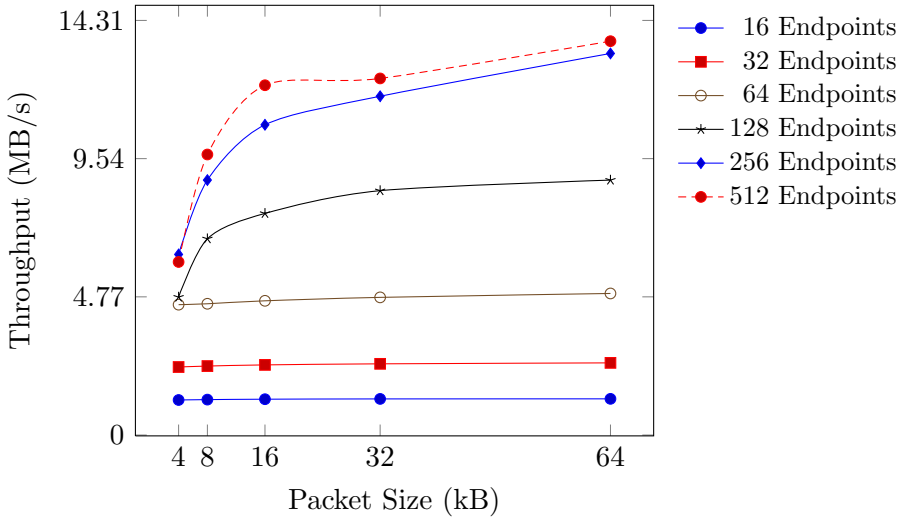


(b)

Figure 5.1: Subfigures (a) and (b) show the effective per packet latency for server-to-endpoint packets and endpoint-to-server packets respectively.



(a)



(b)

Figure 5.2: The throughput in kB/s is shown above. Subfigures (a) and (b) show the server-to-endpoint and endpoint-to-server rates respectively.

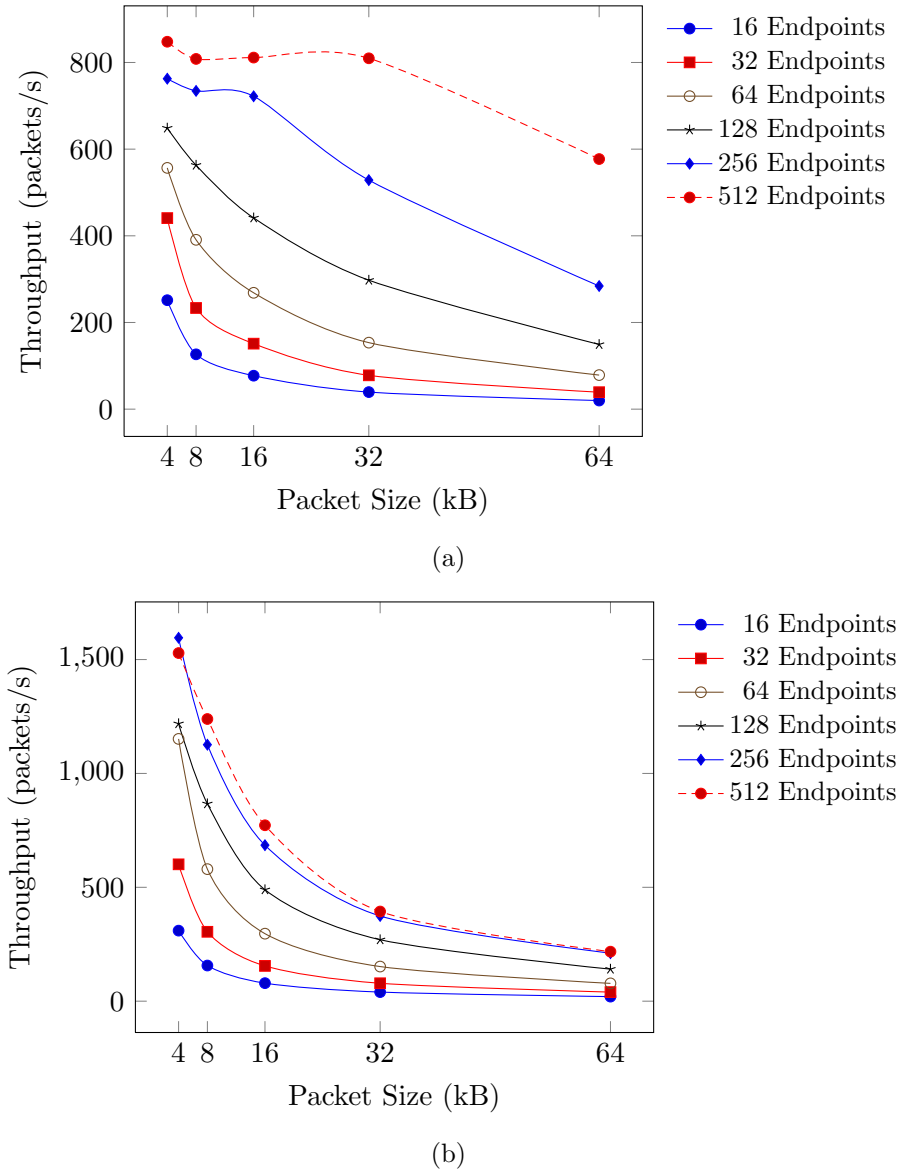


Figure 5.3: Throughput in packets per second is shown for server-to-endpoint and endpoint-to-server packets in subfigures (a) and (b) respectively.

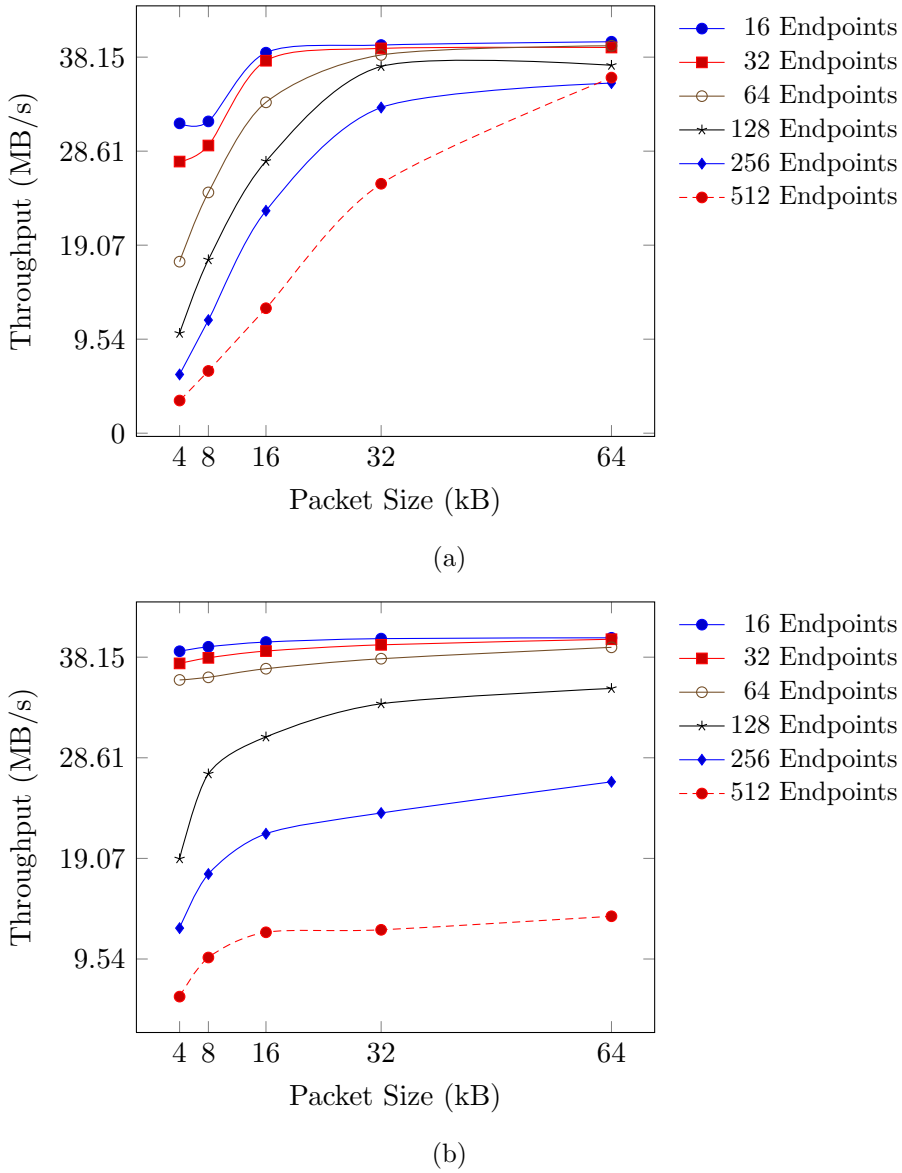


Figure 5.4: The throughput in kB/s is shown above with endpoint scaling. Subfigures (a) and (b) show the server-to-endpoint and endpoint-to-server scaled rates respectively.

5.3.2.5 Security

The authentication phase during handshaking and the use of TLS/SSL as an encapsulation for the protocol, make the system secure. Furthermore, if cloud-based deployment is used, a further level of security is gained.

5.4 Use Case Results

A physical security system use case instance was launched on a single Amazon EC2 Micro instance on 7 August 2012. The instance hosts a single node storage system and a single node data management system. Since its launch and until the time of documentation (a period of 3 months) the system had never failed or denied service.

A total of 170 security endpoints were serviced full-time with a rate limit of 5Kb/s per endpoint. These results finally validate the data management system according to the use case's requirements. Figure 5.5 shows the average CPU usage for 5-18 October 2012.

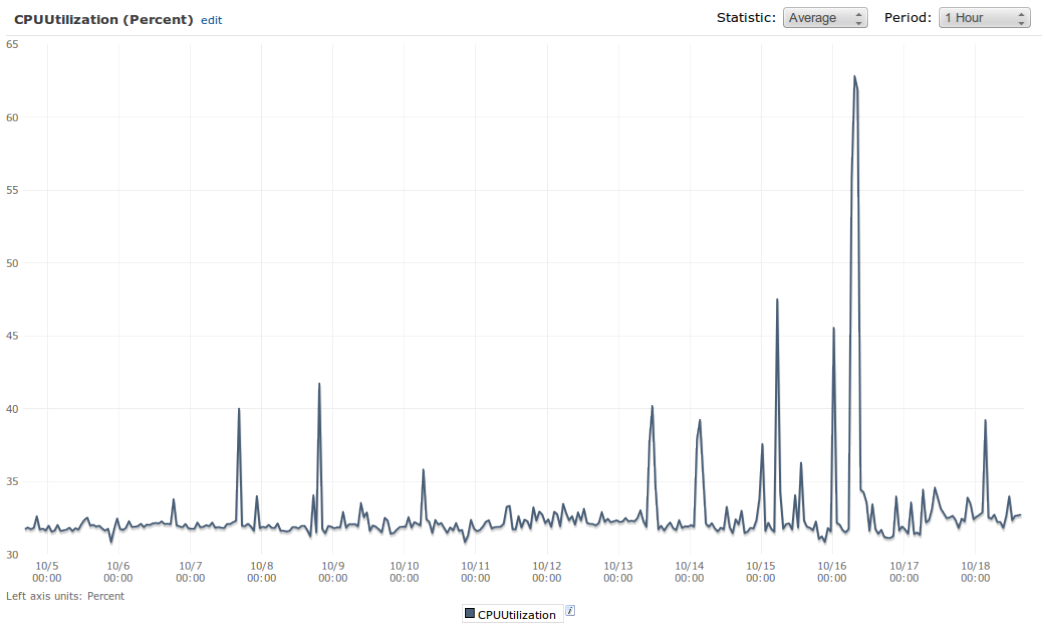


Figure 5.5: EC2 CPU usage.

5.5 Summary

This chapter described the tests performed to verify and validate the data management system. Two separate sets of tests were run to provide verification evidence of functional capability and performance characteristics, as defined in this chapter. The functional capability and performance tests verify the system's functional capability and performance individually. Collectively, the performance tests and practical tests validate the system as a whole.

From the results it is evident that, practically, the synthesized system has met all the functional requirements as shown in table 5.2. The performance characteristics tests showed that the system performs well under significant load. The scalability and availability requirements were shown to have been met by the tests of the system.

A use case instance of the system was practically tested in the field. A single node deployment on a low-end micro instance in Amazon's EC2 demonstrated a capability of servicing 170 endpoints. No errors, performance bottlenecks or loss of services were evident during the 3 month period and it was seen that the processor usage on the instance remained well under 50 % on average.

The empirical tests, together with the use case field test, provide conclusive evidence that the synthesized system is both verified and validated.

A conclusion is the place where you get tired of thinking.

Arthur Bloch

Chapter 6

Conclusion

To conclude this research project, an overview of the functions is provided, followed by a discussion on the final artefact with respect to the data management system's functional and performance requirements. The test results are then reviewed to complete the conclusion that the hypothesis should be accepted.

Chapter 1 introduced the research methodology, namely design science research. A definition of the real-world problem and a description of the use case for the research project were given. A physical security system requires a centralized data management system to provide and manage communication between endpoints. For convenience, the functions of such a system are repeated below:

- Authentication;
- Session support;
- Guaranteed delivery of data packets;
- Segmented packet support;
- Persistent storage of data;
- Packet forwarding;
- Quality-of-service based data billing.

The required constraints of the data management system, as identified by the use case, are listed here for reference:

- Ease of use;

- Scalability;
- Availability;
- Security.

From these requirements, a high-level system architecture was defined that addresses the research problem. The following hypothesis was formulated as a basis for the literature study and synthesis:

A cloud-based implementation can provide the functional capability and performance characteristics for a data management system.

It was shown in Chapter 2 that cloud-based distributed computing is ideal for a scalable and highly-available system. Furthermore, the literature study and preliminary synthesis revealed that MongoDB addresses the requirements for a scalable storage system for this use case.

Nonce authentication was integrated into the system to directly address the authentication and security requirements. I/F 1, which is the protocol defined for the system, utilizes nonce authentication in the handshake phase. The handshaking phase also allows for session re-establishment, which was a requirement.

As the protocol (I/F 1) is the only interface with which endpoints interact, ease of use is only applicable to the protocol. The challenge of consistency lies exclusively with the data management system. The persistence that the storage system provides, allows endpoints to store all states in volatile memory and relies solely upon the protocol and the data management system. Furthermore, no complex key-exchange algorithms are required to implement the protocol. SHA-256 is a simple publicly available checksum tool with trusted source code provided in the standard that defines it.

To further improve usability, all that is required for session renegotiation is to keep track of the last ID and progress of a packet. In the case where an endpoint loses its state, a clean session can be requested. All unconfirmed packets will be re-sent from the first byte. The protocol defines segmented packets and advances the usability further by allowing endpoints to send irregular segment sizes without defining the total size beforehand.

The storage system, that internally makes use of MongoDB, was structured in such a fashion that it can scale infinitely horizontally to address increased load. All states of endpoint sessions are persistently stored in the storage system to serve as a point of synchronisation and consistency. This

point of synchronisation addressed the requirement of guaranteed delivery of packets. Packet payloads and meta-data are persistently stored for billing purposes. QoS defines rate-limits that are used to aggregate and calculate per-volume data billing.

The data management system (F/U 1.0) comprises all the modules needed to securely and consistently service multiple endpoint connections. Interaction between multiple nodes improve the latency of forwarded packets and directly influences the scalability and availability. The shared-nothing principle followed by the system inherently provides infinite scalability and high-availability and directly fits into cloud-based computing. A deployment example for Amazon's EC2 was illustrated in Chapter 4.

A number of recommendations can be made to further improve on the system's capabilities in future research. It was seen that with small packets, 256 message ID's were inefficient. Endpoints were able to queue all 255 messages before receiving the first acknowledgement. By incrementing the message ID's domain to 65535, significant performance gain can be achieved for small packets.

By implementing multiple checksum methods on the system, endpoints can choose different payload verification methods to better suite their requirements. This will require endpoints to provide a list of checksum methods that it is capable of handling.

A web-based front-end module can add functional capability to the system. For example, endpoint data can be extracted from the storage system and can be shown on a web-page as real-time user information. By defining data formats (meta-data) for specific endpoint data packets, customizable *widgets* can be used to better represent the extracted data - for example, geographic data can be displayed on a map, and telemetry data can be represented by dials, gauges, and charts.

Additional protocol packet definitions can provide more services. Typical services that can be added are listed below:

- Tag-based queries of data;
- Online file storage and retrieval per endpoint;
- Online key-value storage per endpoint;
- Administrative functions:
  - Adding endpoints programmatically;
  - Discontinuing endpoints programmatically;

  - Retrieving billing data programmatically;

- The ability to query the online status of another endpoint(s).

In order to verify and validate that this research project solves the research problem, an API was developed and used to test the system in the form of multiple endpoints. The results are now briefly reviewed in order to make a final conclusion.

All functional requirements were tested in functional tests. The successful test results demonstrate that the requirements have been addressed. Performance characteristics were measured for a variety of packet sizes and endpoint concurrency combinations. The performance test results showed that the system scales well with larger packets and also with an increase of concurrent endpoints.

To fully validate the system, the use case was run on Amazon's EC2 on a single Micro instance. 170 actual endpoints had been successfully serviced for a period of over 3 months. The system did not fail or denied service during the test period. This result validates that the requirements have been addressed.

In conclusion, significant evidence is provided that a cloud-based data management system for machine-to-machine communication provides the functional capabilities and performance characteristics required for machine-to-machine communication. The hypothesis is thus accepted for the conditions in which it was verified and validated.

Bibliography

[1] S. March and V. Storey, "Design science in the information systems discipline: An introduction to the special issue on design science research," *MIS Quarterly*, vol. 32(4), pp. 725–730, 2008.

[2] S. Yu, S. Yoon, J. Lee, H. Kim, and J. Song, "Service-oriented issues: Mobility, security, charging and billing management in mobile next generation networks," in *Broadband Convergence Networks, 2006. BcN 2006. The 1st International Workshop on*, 2006, pp. 1 –10.

[3] Y. Tian, B. Song, and E.-N. Huh, "Towards the development of personal cloud computing for mobile thin-clients," in *Information Science and Applications (ICISA), 2011 International Conference on*, April 2011, pp. 1 –5.

[4] D. Welch and M. Shwehdi, "An energy reading and bill generation database for use in nonintrusive load management," in *Transmission and Distribution Conference, 1991., Proceedings of the 1991 IEEE Power Engineering Society*, Sep. 1991, pp. 342 –347.

[5] W. Amer, Y. Attique, A. Nadeem, and A. Ghafoor, "Comprehensive e-monitoring, e-management and e-billing (em2b) system with zoom-in and zoom-out capabilities to reduce electricity distribution losses for developing countries," in *Systems Conference, 2010 4th Annual IEEE*, 2010, pp. 174 –177.

[6] M. Koutsopoulou, A. Kaloxylos, A. Alonistioti, L. Merakos, and K. Kawamura, "Charging, accounting and billing management schemes in mobile telecommunication networks and the internet," *Communications Surveys Tutorials, IEEE*, vol. 6, no. 1, pp. 50 –58, 2004.

[7] T. Choi, C. Kim, S. Yoon, J. Park, B. Lee, H. Kim, H. Chung, and T. Jeong, "Content-aware internet application traffic measurement and analysis," in *Network Operations and Management Symposium, 2004. NOMS 2004. IEEE/IFIP*, vol. 1, april 2004, pp. 511 –524 Vol.1.

[8] X. Wang, B. Wang, and J. Huang, "Cloud computing and its key techniques," in *Computer Science and Automation Engineering (CSAE), 2011 IEEE International Conference on*, vol. 2, june 2011, pp. 404 – 410.

[9] S. Rajan and A. Jairath, "Cloud computing: The fifth generation of computing," in *Communication Systems and Network Technologies (CSNT), 2011 International Conference on*, june 2011, pp. 665 –667.

[10] S. R. Schach, *Object-Oriented & Classical Software Engineering*, 7th ed. McGraw-Hill, 2007.

[11] P. R. C. Coronel, *Database Systems: Design, Implementation, and Management*, B. Woodbury, Ed. Thomson Course Technology, 2007.

[12] MySQL, *MySQL 5.5 Reference Manual*, Oracle, 2011.

[13] B. G. Tudorica and C. Bucur, "A comparison between several nosql databases with comments and notes," in *Roedunet International Conference (RoEduNet), 2011 10th*, june 2011, pp. 1 –5.

[14] N. Leavitt, "Will nosql databases live up to their promise?" *Computer*, vol. 43, no. 2, pp. 12 –14, feb. 2010.

[15] Microsoft. (2010, January) Microsoft SQL Server 2008 R2 - Datasheet. Microsoft. [Online]. Available: http://download.microsoft.com/download/5/5/F/55FA7305-E2A5-485D-81A1-FE7BA2B572F3/SQLServer2008_R2_Datasheet.pdf

[16] K. Simmons. (2010, February) Microsoft SQL Server 2008 R2 Parallel Data Warehouse. Microsoft. [Online]. Available: http://download.microsoft.com/download/4/E/0/4E0ED215-56D8-4D25-9CFC-25F008C73B6C/SQLServer2008_R2_ParallelDW_Datasheet%20v3.pdf

[17] ——. (2010, January) Introducing SQL Server 2008 R2 Datacenter. Microsoft. [Online]. Available: http://download.microsoft.com/download/5/D/5/5D56C9F4-8B57-4215-AB45-E6751DAF1177/SQL_Server_2008_R2_Datacenter_Datasheet.pdf

[18] Oracle. (2009, August) Oracle Partitioning - Datasheet. Oracle. [Online]. Available: http://www.oracle.com/technetwork/database/enterprise-edition/overview/partitioning-11g-datasheet-128345.pdf

[19] ——. (2010, November) Oracle RAC One Node. Oracle. [Online]. Available: http://www.oracle.com/technetwork/database/clustering/overview/ds-oracleraconenode-2009-129387.pdf

[20] D. Gornshtein and B. Tamarkin. (2008, February) Features, strengths and weaknesses comparison between MS SQL 2005 (Yukon) and Oracle 10g databases. [Online]. Available: http://www.wisdomforce.com/resources/docs/MSSQL2005_ORACLE10g_compare.pdf

[21] The PostgreSQL Global Development Group. (2011, September) PostgreSQL 9.1.0 Documentation. The PostgreSQL Global Development Group. [Online]. Available: http://www.postgresql.org/files/documentation/pdf/9.1/postgresql-9.1-A4.pdf

[22] Apache. (2011, August) Cassandra Wiki. Apache. [Online]. Available: http://wiki.apache.org/cassandra/

[23] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with ycsb," in *Proceedings of the 1st ACM symposium on Cloud computing*, ser. SoCC '10. New York, NY, USA: ACM, 2010, pp. 143–154. [Online]. Available: http://doi.acm.org/10.1145/1807128.1807152

[24] MongoDB, *MongoDB Documentation 2011/09/22*, 10gen, September 2011.

[25] Memcached. (2011, September) Memcached Wiki. Memcached. [Online]. Available: http://code.google.com/p/memcached/wiki/NewStart

[26] I. Foster, Y. Zhao, I. Raicu, and S. Lu, "Cloud computing and grid computing 360-degree compared," in *Grid Computing Environments Workshop, 2008. GCE '08*, nov. 2008, pp. 1 –10.

[27] S. Zhang, X. Chen, S. Zhang, and X. Huo, "The comparison between cloud computing and grid computing," in *Computer Application and System Modeling (ICCASM), 2010 International Conference on*, vol. 11, oct. 2010, pp. V11–72 –V11–75.

[28] Amazon. (2012, July) Amazon elastic compute cloud user guide. [Online]. Available: http://awsdocs.s3.amazonaws.com/EC2/latest/ec2-ug.pdf

[29] S. Chuob, M. Pokharel, and J. S. Park, "Modeling and analysis of cloud computing availability based on eucalyptus platform for e-government data center," in *Innovative Mobile and Internet Services in Ubiquitous Computing (IMIS), 2011 Fifth International Conference on*, 30 2011-july 2 2011, pp. 289 –296.

[30] M. Kaitsa, I. Stavrakas, T. Kontogiannis, I. Daradimos, M. Panaousis, and D. Triantis, "Load balancing incoming ip requests across a farm of clustered mysql servers," in *EUROCON, 2007. The International Conference on #34;Computer as a Tool #34;*, September 2007, pp. 546 –550.

[31] R. Aitchison, *Pro DNS and BIND*. Apress, 2005.

[32] S. Oriyano and P. Cabrera. (2008, September) Introduction to Software Load Balancing with Amazon EC2. Amazon. [Online]. Available: http://aws.amazon.com/articles/1639

[33] B. A. Forouzan, *Data Communcations and Networking*, 4th ed. McGraw-Hill, 2007.

[34] K. Raeburn, "Advanced Encryption Standard (AES) Encryption for Kerberos 5," RFC 3962 (Proposed Standard), Internet Engineering Task Force, Feb. 2005. [Online]. Available: http://www.ietf.org/rfc/rfc3962.txt

[35] D. Eastlake 3rd and T. Hansen, "US Secure Hash Algorithms (SHA and SHA-based HMAC and HKDF)," RFC 6234 (Informational), Internet Engineering Task Force, May 2011. [Online]. Available: http://www.ietf.org/rfc/rfc6234.txt

[36] V. Cerf, "Pre-emption," RFC 794, Internet Engineering Task Force, Sep. 1981. [Online]. Available: http://www.ietf.org/rfc/rfc794.txt