**Software Architecture**

**SET10101 Software Architecture Coursework**

*Aaron Campbell*

Table of Contents

[Chapter 1 2](#_Toc57327625)

[1.1 Outline of Brief 2](#_Toc57327626)

[1.2 Distributed Systems 2](#_Toc57327627)

[1.3 Client-Server 3](#_Toc57327628)

[1.4 Peer-to-peer (P2P) 5](#_Toc57327629)

[1.5 Choice of Architecture 7](#_Toc57327630)

[Chapter 2 9](#_Toc57327631)

[2.1 System Design 9](#_Toc57327632)

[2.2 Evaluation 13](#_Toc57327633)

[3. References 15](#_Toc57327634)

# **Chapter 1**

## **1.1 Outline of Brief**

The purpose of this report is to secure a contract for the development and improvement of an ambulance service’s command and control system. The process is to be as follows:

* When an emergency call is received a telephone operator (at organisation HQ) will enter information such as the patient’s name, NHS registration number, address and medical condition to the system.
* The system will check this information against existing records in the database. It will then work out the best way to help the patient and generate an ambulance request to one of 20 hospitals – sending details to the hospital’s computer system.
* The system will then extract the patient’s medical records from the database and send the medical records to a smart phone on the ambulance.
* The hospital will also run a version of the system and will update the patient’s records with the call-out details as soon as the ambulance has entered the details into its smart phone.

The system is expected to be expandable and adaptive to accommodate changing business requirements. The current patient database is accessed via SQL. Eventually it is hoped that the system will interface to the GPS fitted to the rescue vehicles to facilitate location-based functions.

The aim of this report is to identify two software architecture styles which can be used in the design the system, and then make a choice between the two architectures and implement the chosen architecture. A software architecture describes and defines the elements that make up a system, and the relationships among them (Bass et al., 2003).

## **1.2 Distributed Systems**

A distributed system can be described as a series of autonomous computers connected through a communications network. For the purpose of this report, a component refers to processing or data storage (e.g. functions, databases) and connectors refer to data transfer or control mechanisms (inter-process communication, function calls).

Distributed systems offer:

* Economy
* Performance
* Scalability
* Reliability
* Availability

A key aspect of distributed systems is their transparency. This includes end user transparency, developer transparency and designer transparency. The development of distributed systems requires the use of radically different software architectures and components.

However, distributed systems often have more complex functionality than centralised applications – which can act as a challenge in their design. Distributed systems also involve more data between locations.

Communication processes consist of independent communicating through messages. Components may exchange information through a messaging service but do not control each other. They may know the identity of the components with which they are engaging in a dialogue with or not.

## **1.3 Client-Server**

The client-server model is the most common architecture in the development of distributed systems. The first process (the client) issues a request to the second (the server). The server process deals with the request and issues a reply.

A picture containing text, clipart

Description automatically generated

Figure 1. An illustration of the Client/Server architecture.

The architectural design of a client/server program is typically called a communicating processes style (Pressman, 2007). It extends the concept of modular programming to the distributed environment by recognising that those modules do not have to be executed on a single platform (Loosley & Douglas, 1998). The primary benefits of the communicating process style are its distribution for performance and scalability, highly cohesive components and that it is modifiable, portable and reusable. However, due to the nature of its architecture, it must handle unreliable communications.

The *components* of a client-server architecture are the clients and the servers. A server provides a service to an unknown number of clients. The client knows the location of the server and issues a message containing a request. The server performs the action associated with the request and sends a message to the client by way of response. In the client-server model – the client is the service consumer, and the server is the service provider. Clients actively request services from the server, and the server passively waits to serve clients.

Components can be located on any accessible machine. Servers listen to network traffic using an address and a port number. The server may also be managing a number of connections – in which case the server has state information and must keep a record of its connections.

The *connectors*in a client-server architecture at the message-based communications passing data between the components. These messages can be queued and may be lost or delayed. Message transfer error detection and correction must be built into the client-server system. The design of a client-server system includes a consideration of whether the error handling should be built into the connectors or the components.

Connectors can be connection-oriented, which means that a connection is established between the communicating components. The server may be managing a number of connections, in which case the server has state information and must keep a record of its connections.

Client software is typically invoked by a user for one session and runs on that user’s personal computer. Client software generally doesn’t require special hardware or sophisticated OS. It acts as a client intermittently between local processing and can only access one remote server at a time. Server software, however, is generally a special purpose program providing a type of service. Server software is invoked when the OS boots and executes through many sessions, and is run on a powerful, shared computer.

The vast majority of computer based systems exist because software that resides on one computer requests services or data from another computer (Pressman, 2007). Historically, a number of subsystems have been identified that can be present in a client-server system:

* UI/Presentation subsystem – This subsystem implements all functions that are typically associated with the UI.
* Application subsystem – This subsystem implements the requirements defined by the application within the context the application operates.
* Database management subsystem – This subsystem performs the data manipulation and management required by an application.

Another software ‘building block’ exists in all client-server systems, called middleware. Orfali et al. (2007) describe middleware as the nervous system of a client/server system – comprising all software components that exist on both the client and the server and including elements of network operating systems. Once the basic requirements of a client/server system have been formalised, a choice must be made in terms of how software components are distributed across the client and the server. A ‘fat client’ design is when most of the functionality associated with the three subsystems exists on the client side, and a fat server design describes the inverse (Pressman, 2007). Generally, Pressman (2007) offers some guidelines for how components can be distributed:

* The presentation/interaction subsystem is generally placed on the client.
* If the database is to be shared by multiple users, it is typically placed on the server.
* Static data that are used for reference should be allocated on the client.

Alex Berson (1996) was an early contributor to the theory behind the separation of code in a client/server system, breaking the system down into three tiers:

* Presentation functions
* Business logic functions
* Data management functions

The information/bottom tier maintains the data for the application – typically in a database, which may run on a different server called a database server. The middle tier acts as an intermediary and passes requests on to the database. The client/top tier is the application’s UI. This has the advantage of accommodating a ‘thin’ client, which requires less expensive hardware in practice. It also centralises business logic into a single application server and separates it from database functions, making it much easier to maintain.

Diagram

Description automatically generated

Figure 2. An illustration of the three-tier client/server architecture.

Bass et al. (2003) posit that the goal of designing a client/server system is to achieve the quality of scalability. If a server works synchronously in its response to the client, it returns control to the client at the same time as it returns data. However, if it works asynchronously, it returns only the data to the client. Loosley & Douglas (1998) have said that “designing high-performance client/server systems is about making the right connections”.

Colin White (1994) writes that the “benefit of client/server computing is the availability of hardware servers that scale from a small uni-processor machine to a massively parallel machine containing hundreds, possibly thousands of processors... This provides not only scalability, but also flexibility in handling hardware growth as compared with central mainframes where an upgrade is a major undertaking that is both costly and time consuming”.

## **1.4 Peer-to-peer (P2P)**

Peer-to-peer (commonly referred to as P2P) is an architecture that partitions tasks or workloads between nodes. Contrary to client-server, there is no distinguished client or server – instead each node can act as a client or a server depending on whether the node is providing or requesting a service. Each node is a peer.

In the client-server model, the server serves many clients and thus the server must be scaled to the expected workload. In the peer-to-peer model, several computers share resources and communicate in a decentralised way, directly with each other. In effect, peer-to-peer removes the need for a central server.

Diagram

Description automatically generated

Figure 3. The Client/Server architecture.

Diagram

Description automatically generated

Figure 4. The Peer-to-peer architecture.

The *components* in a peer-to-peer system are the peers. Peers can act as both a client and a server and are generally equivalent in functionality.

The *connectors* in a peer-to-peer system are the elements which handle coordination and data exchange. A port describes the messages that the components can process and the messages that it invokes. Connectors interact with the components using elements called roles. Ports and roles are both subject to protocols – set of incoming and outgoing message types, and the valid message exchange sequence (Singh & Haahr, 2006).

Peers typically have similar power and resources as the other peers in the network. Peers share resources, such as processing power, memory and network access. Peer-to-peer’s creation was in large part due to a desire of moving away from centralised control. However, these leads to difficulty in coordinating the peers in an organised manner and thus a detachment of data from sources.

Peers are autonomous, with their behaviours not being under any centralised control - they do not necessarily follow protocol but may be constructed in a way that they favour certain behaviours. Consequently, it is difficult to predict system performance. This also allows for potentially malicious actions being carried out by a peer, bringing security concerns.

A mesh or random swarm network topology is commonly used for a peer-to-peer system – because structured topologies would go against the autonomous nature of peers. A structured architecture can exist at system level, but not at the user level. As a result, it is difficult to provide quality of service guarantees to users. Application level networking is also used among peers, meaning connections can be unreliable. Broken connections are liable to occur, and network links among peers are highly unreliable.

A peer-to-peer system offers excellent scalability and therefore usually scales to an extremely large number of users – meaning it has to be able to handle a large userbase. A peer-to-peer system’s scalability is perpetual - as peers join, resources are aggregated, and more peers are able to join as a result. Therefore, some level of hierarchy has to be used to cope with scalability.

Peer-to-peer networks are typically less secure than a client/server network as security is handled by the individual computers, rather than the network as a whole. The resources of these computers in the network can also become overburdened as they have to support not only their user, but requests from other users on the network.

There are several implementations of the peer-to-peer model:

* Pure P2P – This model is entirely dependent on computers, with no reliance on a central server. Peers find each other dynamically.
* P2P with a simple Discovery Server – A server exists but only to assist peers by providing a list of connected peers. Establishing connection and communication is reliant on peers.
* P2P with a Discovery and Lookup Server – A server (known as a tracker) is used to provide the list of connected peers along with the resources available with each of them.

Several network architectures also exist:

* Structured – Entails a strict set of protocol actions that each node must carry out in order to maintain the network. This architecture is easier to quantify and decide the optimal structures that must be used in the given scenario.
* Unstructured – Peers autonomously decide when and to whom they connect with. This involves random and ad hoc connections among peers. As time goes by, nodes that stay on the network for an extended period of time naturally have a larger number of connections.

Unstructured architectures are currently more popular due to their inherent robustness and higher scalability in comparison to structured architectures.

## **1.5 Choice of Architecture**

For the purpose of this coursework, it was decided to implement a client/server architecture. The primary reason for this is that the main benefits of the client/server architecture are much more conducive to implementing an effective solution for this particular problem than the benefits of the peer-to-peer system – and the potential drawbacks of P2P could potentially be system-breaking.

Peer-to-peer networks are typically less secure than a client/server network. This is because security is handled by each individual computer, rather than the network as a whole. Peer-to-peer’s defining attribute, its autonomous nature, is appealing for some systems but in this case would lead to difficulty in coordinating the peers. This would make it very difficult to predict system performance and could allow for potentially malicious actions from peers – causing glaring security concerns. The standard topologies for peer-to-peer systems are a mesh or random swarm network topology – as these structureless topologies go hand in hand with the autonomous nature of peers. Likewise, it is impossible to offer a structured architecture at user level in a peer-to-peer system. This makes it difficult to make quality of service guarantees to users. Connections can also be unreliable, and broken connections are liable to occur. Nonetheless, peer-to-peer systems offer an unmatched level of scalability in comparison to client/server systems.

Conversely, the benefits of client-server are, in the case of this scenario, its centralised control. Severs aid in administering the entire set-up – access rights and resource allocation is carried out by servers. As a result, centralised backups can also be carried out by servers. Primarily, however, this allows for enhanced security in comparison to peer-to-peer systems. Client/server systems are configured so that access data can be allocated on a per-user basis. Another benefit of a central server is that it is always on – allowing for 24/7 accessibility. With a peer-to-peer system, the computer must always be switched on – which is not practical with client devices that are generally powered off when not in use. It also will not slow down with heavy use and will allow for better client performance, as clients will not have to work as servers. However, server failure leads to the entire network failure and as a result client/server architecture is not as robust as peer-to-peer architecture.

As a result, it was determined that a client/server should be favoured as its benefits would be favoured in terms of the requirements of the system. The primary attributes that must be possessed by the system are:

* Security – The system is handling extremely sensitive information, and so data must be secure and only be given to clients when necessary.
* Reliability – The system must ideally not fail at any time, as this would be catastrophic in terms of its implications on the emergency call.
* Performance – The system must be as fast as possible, as emergency calls are fast-moving.

As described, the client/server architecture inherently has much more robust security over peer-to-peer. It can also generally offer better reliability and performance over peer-to-peer models.

The primary benefit of peer-to-peer, scalability, has no real use in the system. The number of clients is fairly set or, in the future, will possibly be set to a higher number. While the application has a requirement of expandability and adaptability, especially in terms of changing business requirements and a possible larger user base, this level of scalability can be matched by the client/server architecture fairly easily. The level of scalability offered by peer-to-peer systems, while far greater, largely extends to systems where the size of the user base is unknown and is fairly excessive for this application.

# **Chapter 2**

## **2.1 System Design**

For the purpose of this project, Python was chosen as the language with which the system will be developed. The primary reason for this is an existing level of familiarity with Python, as well as its handling of socket programming. Python provides an extensive API that maps directly to the system calls and is a very high-level programming language, meaning a client/server application can be set up in very few lines of code. A brief overview of some of the primary socket API functions and methods are given below:

* socket() – Used in creating a socket.
* accept() – Used on the server-side in accepting a connection from a client.
* bind() – Used to bind the socket to the address (the IP and port).
* connect() – Used on the client-side to establish a connection to the server.
* listen() – Used on the server-side to listen for connections from clients.
* close() – Marking a socket/connection as closed.

Diagram

Description automatically generated

Figure 5. Python API methods and how they provide a client/server architecture.

A high-level description of the system in terms of its components and connectors is as follows:

* The server permanently listens for new connections from clients. The server connects to the database.
* Clients connect to the server.
* The telephone operator client enters patient information.
* The server will receive and broadcast all messages to the necessary parties.
* The server will then determine the best course of action through the patient’s medical records, working out which hospital to send an ambulance request to – if the patient has had a prior ambulance request, a request should be generated to the same hospital.
* A select statement will also be run on the server side, with the query result sent to the ambulance client.
* The hospital is also and a client and will update the patient’s records. The update will be carried out on the server, using an ‘update’ SQL statement.

The flow of messages/*connectors* is as follows:

* The telephone operator enters the patient’s name, NHS registration number, address and medical condition to the server.
* The server receives the message and checks the NHS registration number against records in the database using a select statement. This would be carried out in a method – in Python the NHS registration number could be taken in as a value and then used as a variable in the statement.
* The server generates an ambulance request and sends it to the necessary hospital.
* The server generates a select statement on the patient database and sends the information to the ambulance.
* The ambulance sends the patient call-out details to the server.
* The server sends the call-out details to the hospital.
* The hospital sends updated details to the server.
* The server updates the database using an update statement. This can again be carried out in the same way as earlier statements – the server takes in the message, stores the inputs as variables and passes them in the statement.

The SQL statements would be fairly simple to implement. An example of the SELECT statement is as follows:

|  |
| --- |
| SELECT \*  FROM patients  WHERE reg\_no = “4857773456” |

An example of the UPDATE statements is as follows:

|  |
| --- |
| UPDATE patients  SET address = “10 Park Lane, Edinburgh”  WHERE reg\_no = “4857773456” |

The system is reliant on the server to handle all messages. The server listens for messages from all clients, has methods to deal with these messages and sends messages to the respective clients. It handles database connection for the clients and creates statements for the benefit of the clients. At all stages in the system the server acts as the intermediary between clients – for example, the telephone operator enters information. It is the server who deals with this information and passes it along to the necessary parties.

There are various types of client:

* Telephone operator client – this client enters the patient’s information to the server.
* Hospital client – this client receives call-out details from the server, and then sends updated details to the server.
* Ambulance client – this client receives patient information from the server, and then sends call-out details toe the server.

In terms of the design from the code, inspiration was taken from Berson (1996). The system is largely split, with presentation logic split from the rest of the code. However, it would likely be simpler to describe the system in terms of its subsystems. There is a UI subsystem, implementing all functions associated with the UI such as the creation of labels and buttons and their events, and a database management subsystem, performing the database manipulation and management required. Formally, the *components* of the system can be described as:

* The User Interface
* Telephone Operator Client
* Ambulance Client
* Hospital Client
* Server
* Database

Diagram

Description automatically generated

Figure 6. A high-level view of the KwikMedical architecture.

The user interface is developed using Kivy, a cross-platform Python UI framework. The UI largely takes the form of a chat application, with each client providing a username which would be reflective of their role in the system (for example, Royal Infirmary of Edinburgh). The client enters their initial information (IP, port and username – with their IP read using an in-built socket method) and this information is then stored locally on a text file. Each prior time the client joins the system, their information will be read from the text file. Every time a user sends a message, their username is shown alongside their message.

A picture containing text

Description automatically generated

Figure 7. The main interface of KwikMedical.

The server creates a socket and listens for connections on the socket - and then holds a list of all connected clients. The server has a method for handling received messages, both in terms of accepting connections and in accepting messages from connected clients. When a message is received, it iterates over other sockets in the list of connected sockets and broadcasts the messages. In further iterations of the program, this class would be much larger. For example, the server is where all database manipulation and querying would take place. The server would also require a method for generating ambulance requests and for passing on messages to necessary parties. In its current iteration, the server is capable of taking in messages from a client and broadcasting this to other clients – however, it currently broadcasts the message to all other connected clients. Methods would need to be put in place to ensure that messages (e.g. call-out details, initial information from the telephone operator) are only passed on to necessary clients.

The clients are based off of a relatively simple class, with methods for connecting to the server and sending messages. The connect method connects to the server using Python’s socket API and takes in the client’s username as an input from the user. The class also has a method for sending a message to the server. Finally, the class has a method which constantly listens for incoming messages from the server. It loops over the list of received messages, receives and decodes the incoming message’s according username and then decodes its message content. It then prints any received messages. In future implementations of the system, this class would be unlikely to change very much. This is because of the way the server is expected to work – it should be able to interpret which process the client is acting out. A likely change that would have to be made is having the client select their ‘type’ (i.e. Telephone Operator, Hospital, Ambulance) so that their type can be decoded, and the server can more easily interpret their message.

The database can be implemented using pyodbc, which is a Python module that bridges the gap to OBCD database – compliant databases include MySQL, Oracle and MS SQL Server. This would be handled in another class, or subsystem. This database would contain a table called patients, holding the patients’:

* Name,
* NHS Registration Number,
* Address
* Medical condition;

with their Registration Number being used as the primary key i.e. the value with which a patient can be uniquely identified. All interactions with the database would be carried out on the server side on behalf of the clients.

## **2.2 Evaluation**

There is a lot of scope for the system in its current iteration to be improved. As it stands, the system has a working client/server model, with the server being capable of receiving and dealing with messages i.e. decoding them, determining which client sent them and broadcasting them to other connected clients. However, this currently takes the form of ‘generic’ messages that are simply broadcast to all other clients. In a more concrete iteration of the system, there would be methods in place to deal with these messages in the correct way – depending on which category of message the server is receiving. A more structured system in which messages must be receiving in the ‘correct’ order for an emergency would result in complications, as there will likely be more than one emergency occurring at any given time. However, a system in which a client can determine their type of message or the type of client they are may prove useful.

Nonetheless, the prototype of the system shows that the client/server approach is a viable one and shows signs of some of the attributes described in making the choice to implement the system in this way. For example, the server plays a central role in acting as the ‘hub’ of all operations that can take place in the system – and would allow for 24/7 accessibility in the ‘real’ implementation. This frees the client-side to perform only the necessary actions. The server also shows an ability – which would be dramatically scaled up in a real-world implementation – of coping with several clients and not slowing down in any way, providing immediate responses despite messages having to be decoded through methods before being broadcast. The system is also extremely secure – all data handling is performed on the server side and this would not be affected in any way by any upscaling of the system. These are all attributes which were identified in the initial choice of the client/server architecture (security, reliability and performance) and are clearly present in a small-scale implementation of the system. However, the previously identified issue of a server failure occurring resulting in the failure of the entire system still exists and would have to be accepted as a risk of the design choice.

While the implementation itself is not entirely reflective of the design problem, it offers a good indication of the benefits and attributes of the client/server architecture and proves that the design choices are effective. (talk about actual system design). In terms of the implementation, its codebase is fairly reflective of any new iteration’s architecture. The server acts as the main class of the program, with all client interactions and database manipulation taking place there. This results in a ‘thin’, uncomplicated client design. The system is split into subsystems and tiers, taking guidance from Berson (1996).

The UI subsystem is developed in Kivy, with all UI elements and functionality taking place there. The UI is easily expandable, as its main window is designed in terms of rows and columns and could easily have other elements added to its interface. The current implementation also shows how simply managing screens and swapping between screens is in Kivy, allowing for a fairly simple inclusion of any new necessary screens. One feature that is particular useful is that Kivy is cross-platform and can therefore be packaged and run on Windows, Mac, Linux, Android and iOS. This allows a great level of flexibility in terms of a real-world implementation of the system but, particularly in this problem scenario, would offer a very simple solution in terms of allowing the mobile phone user on the ambulance to use the exact same system as other clients.

The database subsystem does not yet exist but, using pyodbc, could be implemented in a number of ways and catered to user requirements. All data manipulation would take place on the server-side but would, primarily, be stored and accessed via SQL in a similar manner to how the current patient database exists.

The server class takes up the bulk of the code in both the implementation and in the design of the system. This would also be the case in future iterations. Having the system designed in this way means that can be easily expanded and adapted to future needs. Whenever a change must be made to the system, or a feature must be added, this would simply require modification or the addition of new methods in the server class. For example, the real-time tracking of mobile phones is fairly simple in Python using the Google Maps API. Having the system accommodate location-based functions would simply be a case of the rescue vehicles identifying themselves as such and having the server hold a list of all rescue vehicles and their current location, which could be decoded and compared against a patient’s location for faster emergency service. Designing the code in this way and, in general, splitting the code into tiers allows for a simple and easy expansion and improvement of the system in any future iterations.

Overall, it can be said that the client/server architecture is an effective choice of architecture for this problem. The architecture offers undeniable benefits and matches the key requirements set out early in the report: security, performance and reliability. The system is robust in terms of its security and offers almost instantaneous results when performing a request. The design itself is also extremely scalable and can be easily expanded upon with the modification of the server code and, where necessary, the UI code. Put simply, the client/server architecture and the design solution offers un uncomplicated way to improve the system along with the guarantee that security and performance will not suffer as a result.

## **3. References**

Bass, L., Clements, P., & Kazman, R. (2003). *Software architecture in practice*. Addison-Wesley Professional.

Berson, A. (1996). *Client/server architecture*. McGraw-Hill, Inc.

Loosley, C., & Douglas, F. (1998). *High-performance client/server: a guide to building and managing robust distributed systems*. John Wiley & Sons, Inc.

Orfali, R., Harkey, D., & Edwards, J. (2007). *Client/server survival guide*. John Wiley & Sons.

Pressman, R. S. (2007). A Practitioner’s Approach. *Software Engineering*.

Singh, A., & Haahr, M. (2006). A peer-to-peer reference architecture. *2006 1st International Conference on Communication Systems Software & Middleware*, 1–10.

White, C. (1994). Supporting High Performance DSS Applications. *INFODB*, *8*, 27.