I.

2.11.

The simple server is not scalable and can't process many requests all at once. Too many requests would take up all the processing in the unit, as well as stretching the bandwidth of the network. For the server to execute requests within a bounded time, they would either need unlimited processing or the functionality of dropping requests when it is overwhelmed. Both these solutions are not practical, as a server is not reliable when it drops requests and processing is limited.

2.14.

Service A can exhibit serious omission failures through the loss of messages. It is also susceptible to arbitrary failures in the lack of protection for data corruption and possible duplication of messages. These factors combined can contribute to serious reliability issues. The validity is compromised by the loss of messages over the channel, and the integrity is compromised by the possible corruption of data (a lack of any checksum mechanism for the content of the message) and duplicate messages.

Service B can not guarantee validity without some form of message assurance. Since it could drop the message over the channel or experience a receiver-omission failure at the receiver's buffer, Service B would need to use some protocol (like TCP) with acknowledgements to guarantee validity in its message delivery. Service B has integrity since it is not susceptible to duplicated messages and has a checksum for the entire content to ensure the message hasn't been corrupted. Service B, with the right protocol, could be a reliable communication service, but as is it does not have the integrity necessary to be truly reliable.

II.

Middleware is software designed to manage the complexity and heterogeneity of distributed systems. With possible variations in network protocols, operating systems, programming languages, and hardware, Middleware is the adapter that allows distributed applications to run despite these inconsistencies between machines. It is a level of programming abstraction that allows the distributed system to communicate between its parts without needing to worry about the underlying technological differences between its components.

- Distributed Tuples: Middleware used for relational databases. It allows users to apply transactions on a database using a variety of client technologies when interacting with the data.
- Procedure Call: Middleware that allows programmers to invoke a procedure across a network.

- Message-Oriented Middleware: Middleware that abstracts a message queue across various operating systems and the network connecting them.
- Distributed Object Middleware: Middleware that allows clients to invoke methods on remote objects running on other machines.

Game Scenario: Each system can be totally unique, with hardware and operating systems that are not compatible with one another. Say PC1 is running on Windows, PC2 is running on Linux, and PC3 and PC4 are running on MacOS. Middleware allows the games to run on each device without needing to be concerned with the inner workings of each operating system. It allows a level of abstraction above the communication controls used to connect to the server. Say one player is using a keyboard, while another is using a console remote, middleware allows both these input devices to be mapped into the game, giving players the option of variety and giving developers the opportunity to focus only on game mechanics. Middleware would give the game executable independence from the specific system it is running on.

III.

3.1

10 megabits / 1 s = 1,250,000 bytes / 1000 ms
200 * 1000 / 1,250,000 = 0.16 ms to transfer 200 bytes
1000 * 1000 / 1,250,000 = 0.8 ms to transfer 5000 bytes
5000 * 1000 / 1,250,000 = 4 ms to transfer 5000 bytes locally

i. UDP: 5 + 0.16 + 2 + 5 (5 + 0.8) = 36.16 ms

ii. TCP: 5 + 5 + 0.16 + 2 + 5 (5 + 0.8) = 41.16 ms

iii. Local: 5 + 0.16 + 2 + 5 + 4 = 16.16 ms

3.7 ii.

For FTP, it is important to use a TCP to ensure that all the packets get to the destination in order. UDP is prone to packet loss over the network, while TCP uses acknowledgements to guarantee delivery. Receiving and ordering the packets is essential for FTP, since the file will be corrupted if the packets fail at this. TCP is the reliable choice, and FTP requires reliability.

IV. F

13.22.

The layer cake diagram demonstrates the possible organizations of a client-server architecture. It shows that the client application can take up a varying level of responsibility behind the functioning of a program. Some applications require little server contact (perhaps just for the database), while others require help from the server for front end display. For each layer, user interface, application, and database, it is possible for the client and server to share responsibility over the layer. The client can host the entire user-interface software, getting processing and data from the backend, however, it is also possible for the server to dictate remotely how information is displayed on the front end. The client could also be responsible for some of the processing or all of the processing, with the potential to even store its own data.

Examples:

a. Twitter and Facebook
b. Expedia and Google Maps
c. Google Docs and Github
d. Banking apps and Amazon
e. Email (like Gmail) and web browsers

13.23.

A port acts as an endpoint for messages over the network to be routed internally. A machine listens to a specific port for incoming requests. The port is specified in the packet coming in, so the machine knows how to interpret the packet upon its arrival. Below is a list of common port numbers and their usage:

Port	Usage
21	FTP command control
20	FTP data transfer
22	SSH
23	Telnet
53	DNS
80	HTTP
443	HTTPS
143	IMAP
123	NTP
587	SMTP Secure (encrypted email)

V.

Twitter Case Study:

- Transparency: users of Twitter should not be exposed to the underlying complexities behind the application Instead, they should interact with what to them seems like a single, well-oiled machine.
  - Access: Twitter should be accessed in a way that makes it invisible to a naive user that the resources are coming from far away, they should access them fast and reliably.
  - Location: users should not be aware of where the servers running the Twitter service are located. They should be able to access them anywhere with Internet connection without needing to explicitly "find" the resources (DNS works under the hood to find the server's location).
  - Migration: Twitter has massive infrastructure to support all of its users. They should be able to migrate data and processes across servers without the user knowing.
  - Replication: a user should be unaware that their data is replicated across multiple databases to ensure consistency and reliability. Instead, it should seem to them that their data is consistent at all times because there is one true, abstract reality that the data reflects.
  - Failure: a user should not encounter serious network, server, or even client failures to deliver content and exercise Tweets.
  - Concurrency: a user should be unaware that Twitter's servers are supporting millions of requests. Instead, they should always be able to access the website and do what they want to do without any hindrance as a result of busy servers.
- Scalability: Twitter is a massive company that is always scaling up and gaining popularity. The company has to be able to answer millions of requests for service in a timely and reliable fashion. As it scales, it mustn't see a drop in performance or an absurd increase in administrative complexity. Decentralization is a key for a healthy, scalable system. Decentralization decreases the chances of bottlenecks and allows a dynamic state capable of surviving and recovering from failures.
  - Size: as the company grows, it needs to service more users and this requires more resources. Twitter needs servers able to keep up with all the traffic going through them, and it needs to be able to add these servers without hindrance to the overall quality of the distributed system.
  - Geography: Twitter is a multi-national company with users throughout the world. They need to be able to provide access to their content across a wide area, which likely requires duplicate replicas of the service throughout the world. A greater distance results in delays and failures, so it's important to be semi-localized.
  - Administration: as Twitter grows, it requires more technical administrators, and as the system stretches, a poorly scaled system could lead to problems in resource management, financials, and security.

- Dependability: Twitter has a lot behind what goes into delivering and processing client content. At each stage, failures are possible, and small failures can have big consequences if they are not masked properly. Twitter must take measures to protect their system to maintain availability. Dependence can be assisted by load-balancing the requests and having strong enough servers to handle heavy workloads.
- Performance: Twitter operates in a highly competitive market, and if the app ran too slowly or lost information easily, people would migrate to different services. It's important that Twitter stays at the edge of research and development to better the user's experience.
- Flexibility: Twitter's infrastructure must be flexible to allow growth and development, as well as to manage failures and possible problems. Twitter needs the ability to add or replace components to accommodate system growth or change. Twitter needs to set standards for their abstractions to allow integration with some of the many parts of the business.

VI.

4.2

It is important for a client to know the port the service they are trying to contact is running on. If they do not, they may be unable to communicate with the service. If a single machine is running multiple services, then the client can easily get confused if they are using a port tied with another service. The server must clearly publish the ports associated with a service if they want to direct and respond to internet traffic correctly.

The use of a name service or binder can help a client refer to a service by a name, which is then translated to an actual location of the service. If a service is relocated to a different machine, it is important that the binder is consulted to know how it expects to reach the desired endpoint.

4.15

A sender incorporates sequence numbers for their messages before broadcasting them to the multicast group. A sender also requires a unique sender id that they include in every message. Upon receiving a message from a unique sender, the recipient stores the sequence id of the message tied to the sender's id. This lets the recipient know the previous message's number they have received and if the incoming message is next in order. If it is, then they know they are caught up with the sender's messages and have not missed anything, but if there is a gap between the number of the previous message and the number of the incoming message, then the receiver knows they were likely to have missed a message. In this case, for the missing messages, they send a request back to the sender asking for the missing messages, which the sender will then resend.

This scheme ensures that all members actively participating in the group eventually get caught up and miss no messages.