**Parallel Computing**

It is also known as parallel processing. It utilizes several processors. Each of the processors completes the tasks that have been allocated to them. In other words, parallel computing involves performing numerous tasks simultaneously. A shared memory or distributed memory system can be used to assist in parallel computing. All CPUs in shared memory systems share the memory. Memory is shared between the processors in distributed memory systems.

Parallel computing provides numerous advantages. Parallel computing helps to increase the CPU utilization and improve the performance because several processors work simultaneously. Moreover, the failure of one CPU has no impact on the other CPUs' functionality. Furthermore, if one processor needs instructions from another, the CPU might cause latency.

**Advantages**

* It saves time and money because many resources working together cut down on time and costs.
* It may be difficult to resolve larger problems on Serial Computing.
* You can do many things at once using many computing resources.
* Parallel computing is much better than serial computing for modeling, simulating, and comprehending complicated real-world events.

**Disadvantages**

* The multi-core architectures consume a lot of power.
* Parallel solutions are more difficult to implement, debug, and prove right due to the complexity of communication and coordination, and they frequently perform worse than their serial equivalents.

**Distributing Computing**

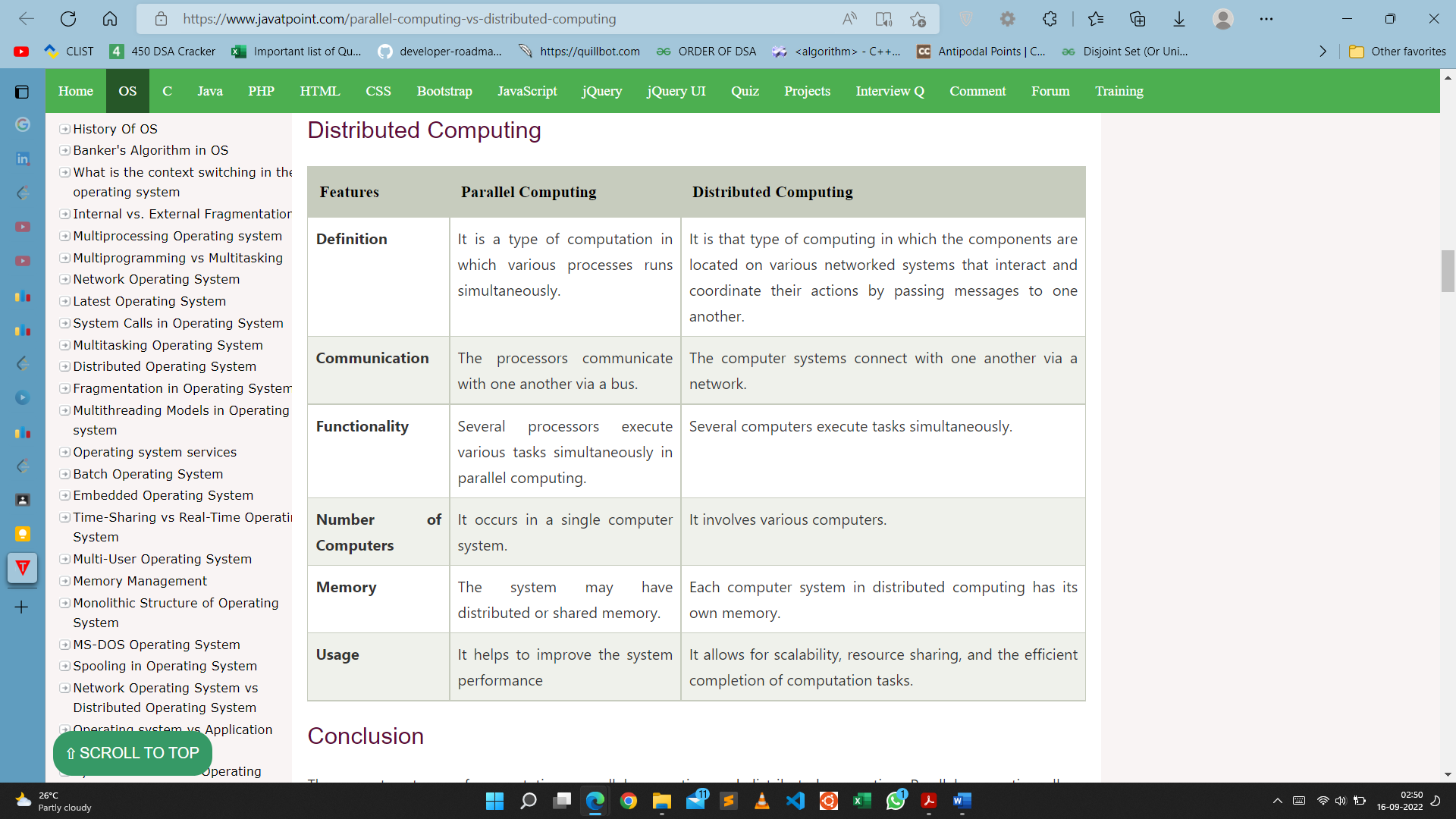
It comprises several software components that reside on different systems but operate as a single system. A distributed system's computers can be physically close together and linked by a local network or geographically distant and linked by a wide area network (WAN). A distributed system can be made up of any number of different configurations, such as mainframes, PCs, workstations, and minicomputers. The main aim of distributed computing is to make a network work as a single computer.

There are various benefits of using distributed computing. It enables scalability and makes it simpler to share resources. It also aids in the efficiency of computation processes.

**Advantages**

* It is flexible, making it simple to install, use, and debug new services.
* In distributed computing, you may add multiple machines as required.
* If the system crashes on one server, that doesn't affect other servers.
* A distributed computer system may combine the computational capacity of several computers, making it faster than traditional systems.

**Disadvantages**

* Data security and sharing are the main issues in distributed systems due to the features of open systems
* Because of the distribution across multiple servers, troubleshooting and diagnostics are more challenging.
* The main disadvantage of distributed computer systems is the lack of software support.

**Shared memory system** is the fundamental model of inter process communication. In a shared memory system, in the address space region the cooperating communicate with each other by establishing the shared memory region.

Shared memory concept works on fastest inter process communication.

If the process wants to initiate the communication and it has some data to share, then establish the shared memory region in its address space.

After that, another process wants to communicate and tries to read the shared data, and must attach itself to the initiating process’s shared address space.

**Message Passing** provides a mechanism to allow processes to communicate and to synchronize their actions without sharing the same address space.

For example − Chat program on the World Wide Web.

Message passing provides two operations which are as follows −

* Send message
* Receive message

Messages sent by a process can be either fixed or variable size. For fixed size messages the system level implementation is straight forward. It makes the task of programming more difficult. The variable sized messages require a more system level implementation but the programming task becomes simpler. If process P1 and P2 want to communicate they need to send a message to and receive a message from each other that means here a communication link exists between them.

Emulating MP over SM:

* I Partition shared address space
* I Send/Receive emulated by writing/reading from special mailbox per pair of processes

Emulating SM over MP:

* I Model each shared object as a process
* I Write to shared object emulated by sending message to owner process for the object
* I Read from shared object emulated by sending query to owner of shared object

RPC

RPC as the primary communication mechanism for distributed applications is due to its following features:

1. Simple call syntax.
2. Familiar semantics (because of its similarity to local procedure calls).
3. Its specification of a well-defined interface. This property is used to support compile-time type checking and automated interface generation.
4. Its ease of use. The clean and simple semantics of a procedure call makes it easier to build distributed computations and to get them right.
5. Its generality. This feature is owing to the fact that in single-machine computations procedure calls are often the most important mechanism for communication between parts of the algorithm [Birrell and Nelson 1984].
6. Its efficiency. Procedure calls are simple enough for communication to be quite rapid.
7. It can be used as an IPC mechanism to communicate between processes on different machines as well as between different processes on the same machine.

By using RPC, programmers of distributed applications avoid the details of the interface with the network. The transport independence of RPC isolates the application from the physical and logical elements of the data communications mechanism and allows the application to use a variety of transports.

**Transparency** RPC is an effective mechanism for building client-server systems that are distributed. RPC enhances the power and ease of programming of the client/server computing concept. A transparent RPC is one in which programmers cannot tell the difference between local and remote procedure calls.

A major issue in the design of an RPC facility is its transparency property. A transparent RPC mechanism is one in which local procedures and remote procedures are (effectively) indistinguishable to programmers. This requires the following two types of transparencies:

1. Syntactic transparency means that a remote procedure call should have exactly the same syntax as a local procedure call.
2. Semantic transparency means that the semantics of a remote procedure call are identical to those of a local procedure call.

Unfortunately, achieving exactly the same semantics for remote procedure calls as for local procedure calls is close to impossible [Tanenbaum and Van Renesse 1988]. This is mainly because of the following differences between remote procedure calls and local procedure calls:

1. Unlike local procedure calls, with remote procedure calls, the called procedure is executed in an address space that is disjoint from the calling program's address space. Due to this reason, the called (remote) procedure cannot have access to any variables or data values in the calling program's environment. Thus in the absence of shared memory, it is meaningless to pass addresses in arguments, making call-by..reference pointers highly unattractive. Similarly, it is meaningless to pass argument values containing pointer structures (e.g., linked lists), since pointers are normally represented by memory addresses.
2. Remote procedure calls are more vulnerable to failure than local procedure calls, since they involve two different processes and possibly a network and two different computers. Therefore programs that make use of remote procedure calls must have the capability of handling even those errors that cannot occur in local procedure calls.
3. Remote procedure calls consume much more time (100-1000 times more) than local procedure calls. This is mainly due to the involvement of a communication network in RPCs. Therefore applications using RPCs must also have the capability to handle the long delays that may possibly occur due to network congestion.

The two types of messages involved in the implementation of an RPC system are as follows:

1. Call messages that are sent by the client to the server for requesting execution of a particular remote procedure
2. Reply messages that are sent by the server to the client for returning the result of remote procedure execution

Call Messages components follows:

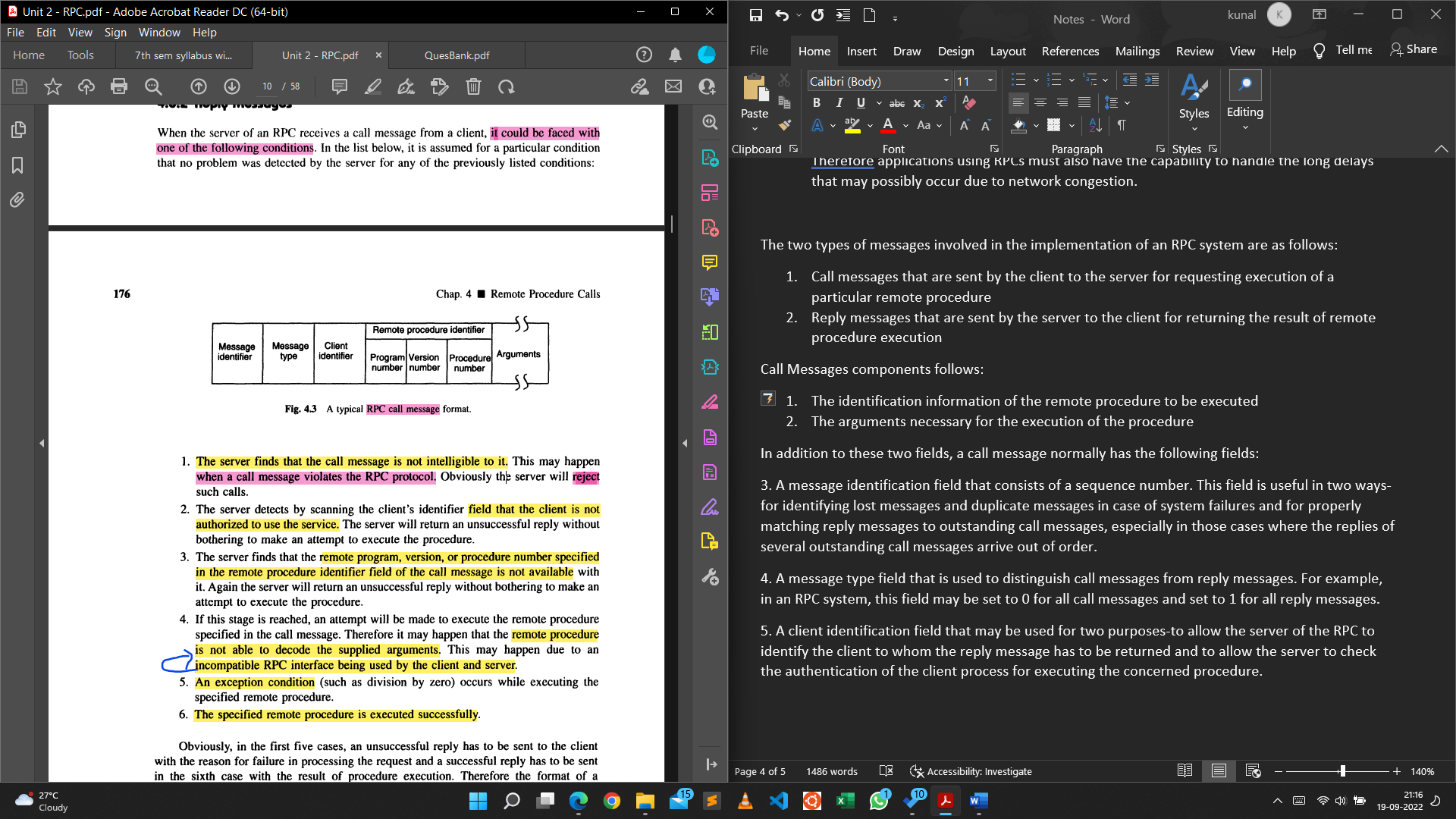
1. The identification information of the remote procedure to be executed
2. The arguments necessary for the execution of the procedure

In addition to these two fields, a call message normally has the following fields:

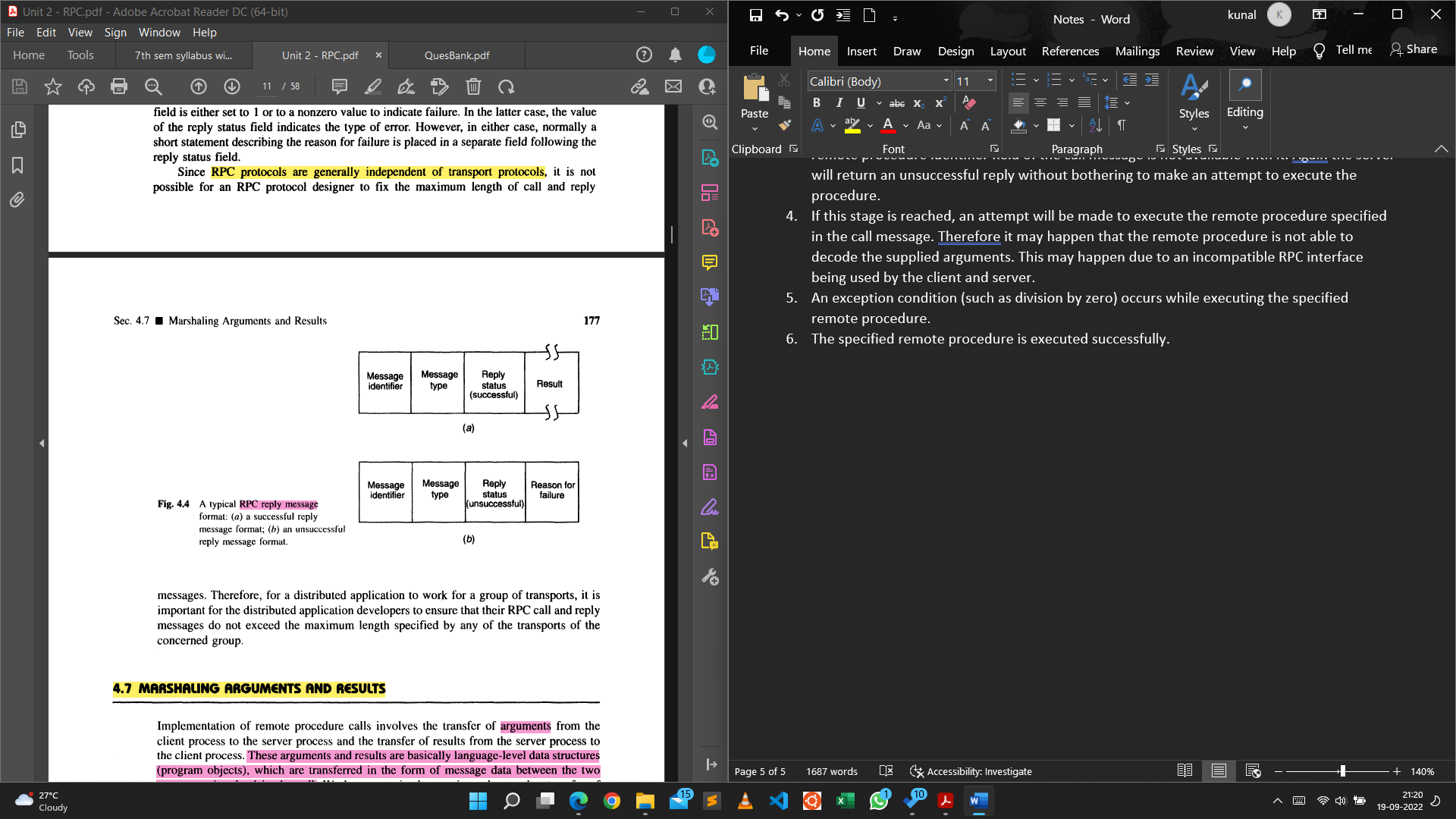
3. A message identification field that consists of a sequence number. This field is useful in two ways-for identifying lost messages and duplicate messages in case of system failures and for properly matching reply messages to outstanding call messages, especially in those cases where the replies of several outstanding call messages arrive out of order.

4. A message type field that is used to distinguish call messages from reply messages. For example, in an RPC system, this field may be set to 0 for all call messages and set to 1 for all reply messages.

5. A client identification field that may be used for two purposes-to allow the server of the RPC to identify the client to whom the reply message has to be returned and to allow the server to check the authentication of the client process for executing the concerned procedure.



It could be faced with one of the following conditions:

1. The server finds that the call message is not intelligible to it. This may happen when a call message violates the RPC protocol. Obviously the server will reject such calls.
2. The server detects by scanning the client's identifier field that the client is not authorized to use the service. The server will return an unsuccessful reply without bothering to make an attempt to execute the procedure.
3. The server finds that the remote program, version, or procedure number specified in the remote procedure identifier field of the call message is not available with it. Again the server will return an unsuccessful reply without bothering to make an attempt to execute the procedure.
4. If this stage is reached, an attempt will be made to execute the remote procedure specified in the call message. Therefore it may happen that the remote procedure is not able to decode the supplied arguments. This may happen due to an incompatible RPC interface being used by the client and server.
5. An exception condition (such as division by zero) occurs while executing the specified remote procedure.
6. The specified remote procedure is executed successfully.

**Marshaling**

Implementation of remote procedure calls involves the transfer of arguments from the client process to the server process and the transfer of results from the server process to the client process. These arguments and results are basicaIJy language-level data structures (program objects), which are transferred in the form of message data between the two computers involved in the call. We have seen in the previous chapter that transfer of message data between two computers requires encoding and decoding of the message data. For RPCs this operation is known as marshaling and basically involves the following actions:

1. Taking the arguments
2. Encoding the message data of step 1
3. Decoding of the message data on receiver’s computer

Marshaling procedures may be classified into two groups:

1. Those provided as a part of the RPC software. Normally marshaling procedures for scalar datatypes, together with procedures to marshal compound types built from the scalar ones, fall in this group.
2. Those that are defined by the users of the RPC system. This group contains marshaling procedures for user-defined data types and data types that include pointers.

**SERVER MANAGEMENT**

In RPC-based applications, two important issues that need to be considered for server management are server implementation and server creation.

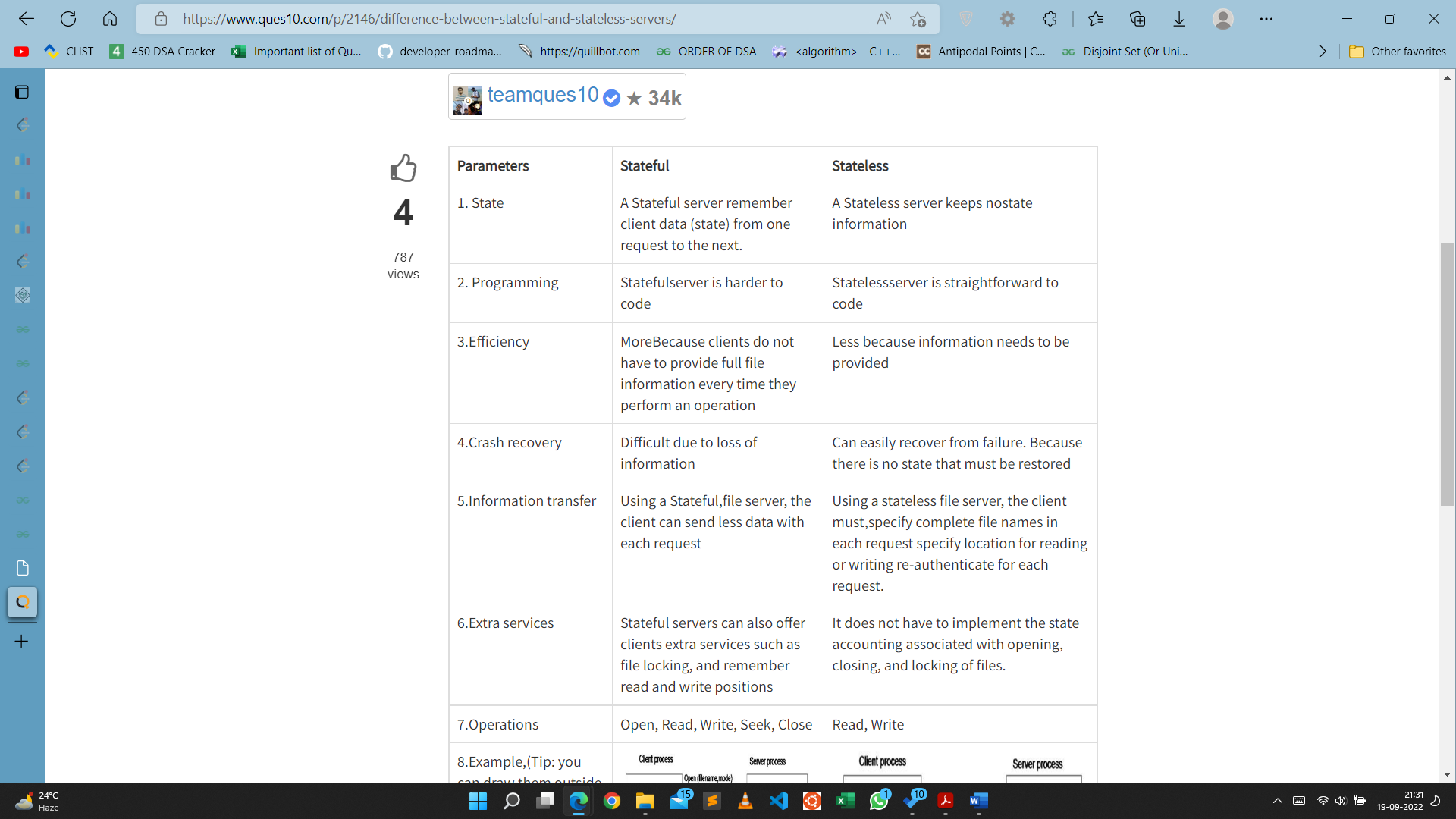
Server Implementation

Based on the style of implementation used, servers may be of two types: stateful and stateless.

Stateful Servers: A stateful server maintains clients' state information from one remote procedure call to the next.

The use of stateless servers in many distributed applications is justified by the fact that stateless servers have a distinct advantage over stateful servers in the event of a failure. For example, with stateful servers, if a server crashes and then restarts, the state information that it was holding may be lost and the client process might continue its task unaware of the crash, producing inconsistent results. Similarly, when a client process crashes and then restarts its task, the server is left holding state information that is no longer valid but cannot easily be withdrawn. Therefore, the client of a stateful server must be properly designed to detect server crashes so that it can perform necessary error handling activities. On the other hand, with stateless servers, a client has to only retry a request until the server responds; it does not need to know that the server has crashed or that the network temporarily went down. Therefore, stateless servers, which can be constructed around repeatable operations, make crash recovery very easy.

Both stateless and stateful servers have their own advantages and disadvantages. The choice of using a stateless or a stateful server is purely application dependent. Therefore, distributed application system designers must carefully examine the positive and negative aspects of both approaches for their applications before making a choice.



**PARAMETER·PASSING SEMANTICS**

Call-by-Value: In the call-by- value method, all parameters are copied into a message that is transmitted from the client to the server through the intervening network. This poses no problems for simple compact types such as integers, counters, small arrays, and so on. However, passing larger data types such as multidimensional arrays, trees, and so on, can consume much time for transmission of data that may not be used. Therefore, this method is not suitable for passing parameters involving voluminous data.

An argument in favour of the high cost incurred in passing large parameters by value is that it forces the users to be aware of the expense of remote procedure calls for large parameter lists.

Call-by-Reference: Most RPC mechanisms use the call-by-value semantics for parameter passing because the client and the server exist in different address spaces, possibly even on different types of machines, so that passing pointers or passing parameters by reference is meaningless.

**Call Semantics**

In RPC the caller and callee processes can be situated on different nodes. The normal functioning of an RPC may get disrupted due to one or more reasons mentioned below:

1. Call message is lost or response message is lost
2. The callee node crashes and is restarted
3. The caller node crashes and is restarted.

In RPC system the call semantics determines how often the remote procedure may be executed under fault conditions. The different types of RPC call semantics are as follows:

1. May-Be Call Semantics

* This is the weakest semantics in which a timeout mechanism is used that prevents the caller from waiting indefinitely for a response from the callee.
* This means that the caller waits until a pre-determined timeout period and then continues to execute.
* Hence this semantics does not guarantee the receipt of call message nor the execution. This semantics is applicable where the response message is less important and applications that operate within a local network with successful transmission of messages.

1. Last-Once Call Semantics

* This call semantics uses the idea of retransmitting the call message based on timeouts until the caller receives a response.
* The call, execution and result of will keep repeating until the result of procedure execution is received by the caller.
* The results of the last executed call are used by the caller, hence it known as last-one semantics.
* Last one semantics can be easily achieved only when two nodes are involved in the RPC, but it is tricky to implement it for nested RPCs and cases by orphan calls.

1. Last-of-Many Call Semantics

* This semantics neglects orphan calls unlike last-once call semantics. Orphan call is one whose caller has expired due to node crash.
* To identify each call, unique call identifiers are used which to neglect orphan calls.
* When a call is repeated, it is assigned to a new call identifier and each response message has a corresponding call identifier.
* A response is accepted only if the call identifier associated with it matches the identifier of the most recent call else it is ignored.

1. At-Least-Once Call Semantics

* This semantics guarantees that the call is executed one or more times but does not specify which results are returned to the caller.
* It can be implemented using timeout based retransmission without considering the orphan calls.

1. Exactly-Once Call Semantics

* This is the strongest and the most desirable call semantics. It eliminates the possibility of a procedure being executed more than once irrespective of the number of retransmitted call.
* The implementation of exactly-once call semantics is based on the use of timeouts, retransmission, call identifiers with the same identifier for repeated calls and a reply cache associated with the callee.