For this discussion board, I'd like to discuss communication and synchronization challenges regarding a distributed computing environment. I personally find cross-machine communication to be interesting and incredibly common in modern development architectures, especially considering how many computing environments take a containerized microservice approach. A lot of distributed environments host their database management systems on machines separate from the virtual machine hosting their applications which also requires concurrency management. There are a lot of complications when considering communication and synchronization in a distributed virtual environment. Just to name a few, network latency, message delivery reliability, concurrency and race conditions between machines, data consistency, load balancing, fault tolerance, and many more contribute to this complexity.

Touching on a few of these, message delivery reliability is incredibly important. Not every message sent over the wire makes its way to its destination, and messages can be delivered to the receiving system out of order or in duplicates. Concurrency and race condition problems also often exist in distributed virtual systems. This usually happens when multiple machines (or instances of a machine) have a shared resource like a shared file bucket or a database server. An example of a race condition would be a process trying to read a database entity, and another process updating a database entity at the same time. These processes could come from different machines, different applications, or different instances of the same application. The final concern that I'd like to further discuss here is fault tolerance. Distributed systems need to be able to handle errors within the processing node, network errors, concurrent operation errors and other faults.

The first concurrency mechanism that I'd like to discuss is message passing. Message passing can take many forms with many protocols. Often in distributed environments messages are passed between machines via HTTP or RPC. Message passing requires two parties, a sender and a receiver, and this sending/receiving process needs some level of synchronization. Synchronization can be achieved in a few different ways, usually through blocking processes on the receiving and/or sending side of the messaging mechanism. This synchronization can be complicated and needs some level of analysis to determine what synchronization method is needed for different scenarios. The three synchronization methods are (Stallings, 2018, p.237):

1. **Blocking Send, Blocking Receive**. Both the sending process and receiving process stop and wait for the messaging mechanism to complete. This is common in tightly synchronized processes; you can see this in synchronous http GET request to an API. Say you have a web application with a page that displays a list of books, and the book data is distributed by an API. The client side will make a request to the API and wait for a response from the API so it can build that display.

2. **Nonblocking Send, Blocking Receive**. This is also a common messaging mechanism, where the sender doesn't care what happens after it sends the message. This is also prevalent in web development. Imagine in this web application, a book is removed from the list. If the client application doesn't care to display a response to the user, it can send a request to the API to remove the book from the database and continue being responsive to the user. The API is blocked until it receives a message, so it's patiently waiting for messaging coming from the client application to do work. That's the "Blocking Receive" part of this synchronization mechanism.

3. **Nonblocking Send, Nonblocking Receive**. In this case, neither the sender nor receiver is waiting to process other messages. You can see this in an event driven architecture, where the sender might send an event to any listening service. That listening service might be operating on a queue of messages, and it'll operate on the message distributed by the sender when it's that messages turn. The sender won't wait for a response, and the receiver isn't necessarily waiting for a message. A production example of this would be a claims system. A claim might be filed by a claimant, which will send a message to a message broker. The broker will distribute the message to a service that evaluates the claim and determines potential benefits for the claimant, and another service that determines whether the claimant is qualified for benefits. The initial sender doesn't care for a response, and the subscribing services might be working on operations from other messages. Eventually when all operations are completed on the initial claim, correspondence might be sent to the claimant from a fourth service via email, but there would not be immediate confirmation that the claimant does or does not qualify for benefits.

For each of these synchronization methods I provided real world scenarios where they might be used to resolve synchronization complications. This becomes increasingly complex when considering message reliability, message security, load balanced systems and parallel operations between duplicate application nodes. There are some tools that attempt to provide guarantees in message ordering to help with this like Apache Kafka (a common message broker in event driven architectures), but most of the time reliability needs to be built into the receiving application using idempotent development practices and data modelling, event sourcing, versioning, and compensating transactions. These are somewhat out of scope of this assignment, but it's something that I've been studying for work.

Another concurrency mechanism that can often be found in distributed virtual environments is a Mutex or Distributed lock. The term Mutex is short for "Mutual Exclusion" and is used to ensure that multiple threads or processes don't access a shared resource at the same time. If a mutex isn't used in scenarios where multiple threads or processes are going to use a shared resource, this could result in synchronization issues like race conditions. Mutexes protect against race conditions by locking a region of code or resources prior to operation so that another thread or process that wants to use that code or resource must wait until the lock is released. (Mutex lock for Linux Thread Synchronization – GeeksforGeeks, 2017). In distributed mutual exclusion, this shared resource is often called a "Critical Section".

Distributed systems don't share memory or a physical clock, so mutual exclusion synchronization is harder to achieve and usually requires message passing. GeeksforGeeks provides some good information on algorithms to ensure distributed mutual exclusion (Mutual exclusion in distributed system, 2019):

1. **Token Based Algorithm**. Each service that needs access to the critical section has the possibility of possessing a token. The token is required to enter the critical section. Once the process completes its operation in the critical section, it passes the token on to another service that requires access to the critical section.

2. **Non-token Based Approach**. The process attempts to enter the critical section via message passing. The message with the earliest time stamp gains entry first. The queue of access requests is sorted by request time stamp.

3. **Quorum Based Approach**. The process requests permission to access the critical section from multiple groups (quorums) of machines. If a specified number of quorums grant the process access to the critical section, the process can enter the critical section.

Since we're operating in a distributed virtual environment, we run into all the same communication problems that come from message passing since that's going to be the main mode of communication between processes. (Mutual exclusion in distributed system, 2019) (Stallings, 2018, p. 18-12)

Stallings, W. (2018). Operating Systems: Internals and Design Principles. Pearson Education Inc.

Mutex lock for Linux Thread Synchronization - GeeksforGeeks. (2017, June 2). GeeksforGeeks. https://www.geeksforgeeks.org/mutex-lock-for-linux-thread-synchronization/

Mutual exclusion in distributed system. (2019, April 25). GeeksforGeeks. https://www.geeksforgeeks.org/mutual-exclusion-in-distributed-system/