Question 1:

1. Middleware: a software layer that act to provide abstraction and hide the heterogeneity of the network (different hardware, OS, and programming languages). Act as a bridge between the OS and the network of an application.
2. An unexpected random behaviour of a process that is supposed to run an algorithm, due to: faulty components, unreliable network, insecure network.
3. During commit-voting phase:  
   An elected coordinator attempt to query each component by asking it commit?  
   And expects a response of commit or abort.  
   If all components commit, it sends a global commit to perform the action.  
   If any component respond with abort, it sends a global\_abort to revert back to previous stage.
4. It means it can cope with at-least once model which state that the sender will send a message (multiple time if needed) and recipient will eventually receive at least one copy of the message. Meaning it has anti-duplication mechanism and fail-storage
5. Because that would require no latency between clocks, and this is impossible in a distributed system (bottlenecks includes: connection and limitation of speed of light, processing speed, material used in cabels)
6. A registry of all methods supported by the service, so the client can query them using strings.
7. Packing parameters into a message within RPC, usually into a stub. Used to counter the heterogeneity of distributed systems.
8. Replication: used mainly for fault tolerance as well as performance. Keep constant updates (eventually). Expensive.  
   Caching: Reduce traffic load on servers, brings copies closer to servers if content is being requested a lot. Reactive to usage; we only cache what is being requested not cache beforehand (or what we expect to be used a lot). Cheap.
9. Little’s law used in simulations to tell us:  
   The average number of requests in a queue based on the arrival rate multiplied by average process time.   
   L = λW
10. Send multiple requests that I don’t care about their results, in quick succession without delay by using as many threads/computers I can. This will deny the computing resources to other people and slow down the server as it will be busy dealing with my meaningless requests.

Question 2:

1. Because if the proportion of the application that can be parallelised (x) is small then the speedup will be also small. Calculated by speedup = 1/(1-x) and that is not all, introducing threads has costs associated with it due to overhead of communication/synchronisation so it might end up slowing the system!  
   Amdahl’s law: used to find the maximum expected improvement to an overall system when only part of it is improved. Used to predict the theoretical speedup in parallel computing.
2. ACID:  
   Atomic: All or nothing; either all operation committed successfully or none committed.   
   Consistent: Every transaction leave the system in a consistent state respecting the system integrity.  
   Isolated: Components don’t interfere with each other, (after concurrent operation finish execution results are like they have done serially)  
   Durable: Updates are persistent once they are committed.
3. 1. Availability = 1 - p(failure)  
      p(failure) = (0.5/2.5) \* (0.5/4) \* (0.5/12.5) = 0.001  
      Availability = 0.999  
      Meaning the service is highly available.
   2. Availability = 1 - p(failure)  
      p(failure) = (0.5/2.5) \* (0.5/4) = 0.025  
      Availability = 0.975  
      Meaning the service is highly available.
   3. I would choose whatever maximise the mean time between failures, and minimise the time required to fix a failure.
4. Lamport clocks:  
   P1 - 3  
   P2 - 5  
   P3 - 4  
   P4 - 4  
   Vector (from left to right)  
   P1 - (0, 0, 0, 1) < (0, 0, 0, 2) < (0, 0, 0, 3)  
   P2 - (1, 0, 1, 0) < (1, 2, 2, 1) < (1, 2, 3, 2)  
   P3 - (0, 1, 0, 1) < (0, 2, 0, 1) < (2, 3, 0, 1)  
   P4 - (1, 0, 0, 0) < (2, 0, 0, 0) < (3, 0, 0, 3)

Question 3:

1. Because network has latency, there are a lot of bottleneck factors such as noise introduced by cables that are used to transmit data, speed of light being a limiting factor as data usually get transmitted across the globe and so on .
2. Client that wants to invoke a remote procedure first calls the client stub to marshal the arguments. The stub main job is to provide an abstraction and hide the heterogeneity of the distributed system. After marshalling the arguments the client does OS call to send the stub across the network to the server. The server does a server stub call to unmarshall the stub and get the argument and does its computation. Then it calls server stub again to marshall the result, send it across the network, the client calls the client stub to unmarshall and get the results.
3. - Process sends an election message to all processes with higher identifier than itself.  
   - Those that haven’t crashed will respond with an answer (within timeout) - initiating process can then sit back and wait for a coordinator message   
   - any process receive an election message should respond back to the sender.  
   - And then start its own election process   
   - unless it has recently done that and haven’t had a reply yet.   
   - The process with no answering process will consider itself elected  
   - It should then send a coordinator message to all lower processes
4. 1. Not possible - look at past paper 2015 for explanation. Basically P3 and P4 will want different order of reads to be done.
   2. Yes.  
      P1 - P1(x) = 2  
      P2 - P2(x) = 1  
      P3 - pass while  
      P3 - y = P2(x) // y = 1  
      P3 - y = 4 \* y + P1(x) // y = 4 \* 1 + 2  
      P1 - P1(x) = 3  
      P4 - pass while  
      P4 - z = P1(x) // z = 3  
      P4 - z = 4 \* z + P2(x) // 4 \* 3 + 1

Question 4:

1. Because no way to know for sure if the message reached the other general; it could be lost to the enemy. The other general might receive the message but the first general get skeptical if the other one received or not, the only way to somewhat have a good confidence that the message is reached is sending it as many time as possible this will increase the likelihood that the message will reach safety, but never a 100%.
2. Byzantine Generals problem - rather long to answer this just check lecture 15.
3. Components can fail all the time, network isn’t secure nor reliable so we need to be able to handle that or at least amend as much as possible. When component is giving faulty results we need to be able to identify it and rule it out, we need to know whtat messages won’t always reach the other system 100% all the time so we ensure we send enough messages to get as close to 100% as possible.
4. 1. Total server ability = 26 + 18 + 56 = 100  
      To load balance: Server X / Total Server \* number of request  
      A - 18 / 100 \* 300 = 54  
      B - 26 / 100 \* 300 = 78  
      C - 56 / 100 \* 300 = 168
   2. We can handle 100 request per second (Server A + B + C)  
      If we have 50 clients, then at most they each can make 2 requests, else further requests will be placed in the queue
   3. To each server S  
      Total server ability = sigma sum of all si from i = 0 to n   
      Allocation to S = s / Total server ability \* x