

# Assignment Project Exam Help

Distributed System Models

<https://powcoder.com>

Aaron Harwood

School of Computing and Information Systems

© The University of Melbourne

Add WeChat powcoder

2022 Semester II

## 1 Modelling Overview

### 2 Functional Models

- Processes and Machines
- Common architectural patterns
- Network characteristics
- Distributed system topology
- System state and overheads

### 3 Non-functional Models

- Temporal Model
- Failure Model
- Security Model

Assignment Project Exam Help

<https://powcoder.com>

Add WeChat powcoder

## Functional and non-functional models

In this section we will *develop a model* of a distributed system, from first principles.

- Developing the model means specifying a notation for representing a real-world distributed system and its environment as an ideal (read mathematical) object or concept.
- Functional models are concerned with how the distributed system implements functionality, the tangible aspects of the distributed system. In this section our Functional models will be *time-independent* - that is they describe aspects of the system that are constant over the system's lifetime.
- Non-functional models are concerned with aspects which are intangible such as time, reliability and security. When we model time then we can revise our Functional models to provide time-dependent Functional models.

In this section we will demonstrate how the models can be used to make some basic statements of a distributed system's properties and behaviour.

## Processes and machines

# Assignment Project Exam Help

The *Architectural Model* is, in a broad sense, concerned with the components or entities of the system, the roles that they take, the relationships between them, and how they are expected to interact with each other. In this subject we are satisfied by the components being considered simply as processes, and to describe the physical hardware/OS, i.e. the platform or platforms that the processes execute on.

- *Architectural elements* – components of the system that interact with one another
- *Architectural patterns* – the way components are mapped to the underlying system
- *Associated middleware solutions* – existing solutions to common problems

## Process Roles

Each process has a high-level purpose or role that it takes in the distributed system. Here are some common examples of process roles:

- User Interface Process – a process that allows users to interact with the distributed system, either graphical or command line based. It usually acts as a client to other processes in the distributed system.
- Data Cache – a process that caches information, typically in memory, for fast access by other processes.
- Communication Relay – a process that other processes can connect to in order to undertake indirect communication with other processes.
- Job Manager – a process that can start, stop and monitor other distributed system processes.
- Index or Registry Server – a process that provides the location, e.g. the network address, of resources in the distributed system.
- Database Server – a process that provides a database service. A database server process allows other processes acting as database clients to connect to it and obtain database services.
- Authentication Server – a process that authenticates users, e.g. by passwords or other security mechanisms. Other processes in the distributed system may require users to authenticate with the Authentication Server prior to providing service.

## Middleware

In most cases the roles described earlier can be undertaken by a group of processes that we sometimes refer to as a sub-system e.g.:

- Reliable Key-Value Storage – a fault tolerant general purpose storage system
- Cluster Resource Management – helps to manage resources on a cluster of machines
- Enterprise User Management – helps to manage users in a large organization
- High Performance Messaging – provides optimized communication mechanisms

They are usually referred to as *middleware*. As part of our distributed system architecture we may choose to use various kinds of middleware as sub-systems that solve common problems for us in a distributed way. We should understand the architecture of the sub-system and what impacts it has on our distributed system architecture as a whole. Our models help us do this.

## Machine Roles

Similarly to processes, machines tends to have well defined roles in the distributed system, and here are some examples:

- Desktop/PC – runs client processes, has a fixed location at home or at work, user may or may not be at the machine, has no power limitations, has reasonably good resource capacity, usually connected to the network via Ethernet, may be switched on for long periods of time, may or may not have a public IP address
- Mobile Device – runs client processes, is mobile – may connect via all kinds of public network, tends to be with the user 24/7, has power limitation, usually limited resource capacity, only connected via wireless WiFi or cellular, may be on 24/7, usually won't have a public IP address
- Front End Server – runs server processes that allow clients to connect to it, located in a machine room at a fixed location, is only remotely connected to, has no power limitations, has significant resource capacity, connected via high speed Ethernet, on 24/7, has a public IP address
- Back End Server – runs distributed processes to support front end servers, located in a machine room at a fixed location, is not reachable from the public Internet, has no power limitations, has significant compute and storage capacity, connected via high speed Ethernet, on 24/7, has a private address
- Virtual Machine/Server – runs on cloud infrastructure, provides capabilities similar to Front/Back End Servers.

## Interfaces and IPC mechanisms

- Distributed processes can not directly access each others internal variables or procedures. Passing parameters needs to be reconsidered, in particular, call by reference is not supported as address spaces are not the same between distributed processes. This leads to the notion of an *interface*. The set of functions that can be invoked by external processes is specified by one or more *interface definitions*.
  - Programmers are only concerned with the abstraction offered by the interface, they are not aware of the implementation details.
  - Programmers also need not know the programming language or underlying platform used to implement the service.
  - So long as the interface does not change (or that changes are backwards compatible), the service implementation can change transparently.
- Choices of IPC mechanism include:
  - Direct IPC are the underlying primitives – relatively low-level (perhaps the lowest level) of support for communication between processes in a distributed system, e.g. shared memory, sockets, multicast communication
  - Remote invocation – based on a two-way exchange between communicating entities in a distributed system and resulting in the calling of a remote operation, procedure or method, e.g. request-reply protocols, remote procedure calls, remote method invocation
  - Indirect communication:
    - *space uncoupling* – senders do not need to know who they are sending to
    - *time uncoupling* – senders and receivers do not need to exist at the same time
    - for example: group communication, publish-subscribe systems, message queues, tuple spaces, distributed shared memory



## Placement

Placement of processes onto machines in the distributed system is a fairly central aspect of the distributed system architecture and design problem. E.g.:

- mapping services to multiple servers – a single service may not make use of a single process and multiple processes may be distributed across multiple machines
- caching – storing data at places that are typically closer to the client or whereby subsequent accesses to the same data will take less time
- mobile code – transferring the code to the location that is most efficient, e.g. running a complex query on the same machine that stores the data, rather than pulling all data to the machine that initiated the query
- mobile agents – code and data together, e.g. used to install and maintain software on a users computer, the agent continues to check for updates in the background

<https://powcoder.com>

Add WeChat powcoder

## Modelling processes, machines and process placement

We can start modelling a distributed system by consider the collection of processes, or let us say  $n$  processes in order to model it.

Similarly we can consider the collection of machines that these processes will be distributed over, let us say  $m$  machines in this case.

Each process runs on a machine. If we wanted to model this we could define a mapping or placement from processes to machines. Let's say if  $i \in \mathcal{N} = \{1, 2, \dots, n\}$  is a process, referred to as process  $i$ , then let  $P_i = j$  be the machine  $j \in \mathcal{M} = \{1, 2, \dots, m\}$  that process  $i$  is placed on.

We could simplify the model by assuming that there are as many machines as there are processes,  $m = n$ . Some distributed systems can be modelled this way, e.g. peer-to-peer file sharing systems tend to have a peer running on each user's computer. There are no other processes or machines involved.

## Process resource requirements and machine capacity I

Lets continue to expand the model. Let  $L_j$  be the number of processes that are placed on machine  $j$ :

$$L_j = |\{i \mid P_i = j\}|$$

Load can be interpreted as a simple or *first order measure* of the resource capacity requirements of a machine – every process requires resources; cpu, memory, network, storage.

We can model each processes' resource requirements if we wish and we can also model each machine's capacity. A more detailed model becomes harder to understand but it allows us to make more precise statements about the distributed system. Let's see where we can get to in this direction.

We could simply say that each machine has an integral capacity of  $K \geq 1$ , meaning that no more than  $K$  processes can be placed on any single

## Process resource requirements and machine capacity II

machine. Our machines are said to be *homogeneous* with respect to capacity. Or we could allow our machines to be *heterogeneous* and model the capacity as  $C_j \in \mathbb{Z}^+$ , i.e.  $C_j > 0$  is an integer. The case when  $C_j = K$  for all  $j$  is the (special) homogeneous case of the heterogeneous model.

If we like we could have used the positive real numbers  $\mathbb{R}^{>0}$  for our model instead of the positive integers  $\mathbb{Z}^+$ . Doing so may be somewhat more realistic – a given machine's capacity may well be e.g. 5.5 meaning that when executing 5 processes the machine still has spare capacity but not enough to support 6 processes. In engineering analysis and optimization we may want to use real numbers, but for the analysis and optimization of distributed algorithms and when analysing a distributed system's architectural properties we may prefer integers.

We could model each of the machine's primary resource capacities – cpu, memory, network and storage – with more detail. If this is desired we

## Process resource requirements and machine capacity III

could write  $C_{j,\text{resource}}$  to represent the capacity of a specific resource on machine  $j$ , e.g.  $C_{j,\text{cpu}}$  could be the number of cpus on machine  $j$ . Or more succinctly  $C_{j,r}$  could refer to the capacity of resource  $r \in \mathcal{R}$ , which is some set of resources that we are interested in. We generally do not need to model machine resources beyond a simple first order measure, but we can if we wish.

When modelling the network resource we tend to talk in terms of available upload (data going out of the machine) and download (data coming into the machine) bitrates. The network is an important resource to model because all of our IPC will make use of it. There are other considerations as well with respect to network modelling since it is impacted by the computer network that our machine is connected to, and as well the other machine in the communication. We will specifically revisit our model of the network later.

## Process resource requirements and machine capacity IV

On the other hand we have process requirements. It is implicit in the model that each process requires to be placed on a machine. However continuing our definition of resources we could let  $P_{i,r}$  be the amount of resource  $r$  that process  $i$  requires. This could be considered the maximum amount of that resource that the process will consume when running. With these basic definitions in place we can consider the total resource requirements for resource  $r$  on a given machine  $j$ :

$$L_{j,r} = \sum_{i \mid P_i=j} P_{i,r}$$

## Analysis: load balancing and placement problems I

We can now formulate a number of interesting statements and problems with respect to process placement on machines and resource consumption. E.g. we might want to assert that no machine is allowed to execute more processes than it has capacity for:

<https://powcoder.com>

In other words we might assert that the placement of processes should never exceed the resource capacity of any machine. We might indeed formulate the problem of finding a placement of processes that achieves some objective (subject to the capacity restriction above), e.g. minimizing the maximum consumption of resource  $r$  over all machines,

$$\arg \min_{\{P_1, P_2, \dots, P_n\}} \max_j \{L_{j,r}\},$$

## Analysis: load balancing and placement problems II

minimizing the average percentage resource consumption over all machines,

$$\arg \min_{\{P_1, P_2, \dots, P_n\}} \max_j \left\{ \frac{1}{|\{\mathcal{R}\}|} \sum_r \frac{L_{j,r}}{C_{j,r}} \right\},$$

or finding a placement that minimizes the number of machines that are required (bin packing problem) to satisfy the total process requirements:

$$\arg \min_{\{P_1, P_2, \dots, P_n\}} m,$$

where in this case  $m = \max\{P_i\}$  and again, the placement is subject to the capacity restriction given earlier. The bin packing problem is strongly **NP** complete.



## Connections between processes

Interprocess communication is an essential aspect of the Architectural Model. We can expand our model to include the notion of IPC by defining communication as a *directed edge* between two processes  $i, i' \in \mathcal{N}$  which we can write as the tuple  $(i, i')$ , and then define the set of all connections:

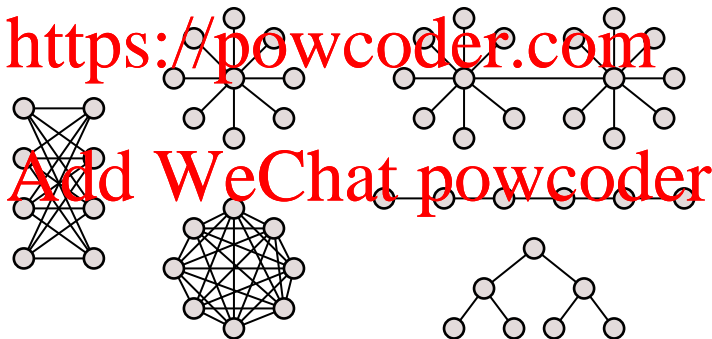
$(i, i') \in \mathcal{E}_n$  if process  $i$  can initiate communication with process  $i'$

To be specific, we can say that if  $(i, i') \in \mathcal{E}_n$ , then process  $i$  can initiate communication with process  $i'$  (i.e. using TCP or UDP) at least once in the lifetime of the distributed system (the model is currently not specifying exactly when or even if it does, just that it could take place). If for two processes  $a, b \in \mathcal{N}$  those processes can never directly communicate in the lifetime of the distributed system then  $(a, b) \notin \mathcal{E}_n$  and also  $(b, a) \notin \mathcal{E}_n$ .

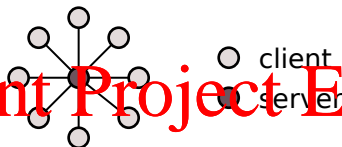
Sometimes we use undirected edges, written  $\{i, i'\}$  when its not important to consider which process initiates the communication, or when either process could initiate the communication. With this model,  $\{i, i'\} \in \mathcal{E}_n$  is equivalent to  $(i, i'), (i', i) \in \mathcal{E}_n$ .

## Example process connection patterns

The connection “pattern” or *architecture* that arises between processes is an essential aspect to model and understand in a distributed system – we are usually especially interested to understand such patterns as the number of processes in the system grows larger, or to the extreme as  $n \rightarrow \infty$ .



## Centralized Pattern




# Assignment Project Exam Help

Client processes communicate with a Server process. Clients never directly communicate with each other. This pattern is the dominant design pattern for distributed systems:

- It is relatively easy to implement.
- All data is kept at the server.
- If the server fails or is attacked, the distributed system fails – single point of failure.
- Since clients never directly communicate, they are not exposed to each other over the network – privacy/security.
- If some clients fail or are attacked, it does not have an impact on other clients.
- A single server may not be able to support a large number of clients – scalability bottleneck.
- Depending on the geographic/network location of the server to the clients, different clients may see different performance characteristics – latency may vary among clients.

## Decentralized Pattern

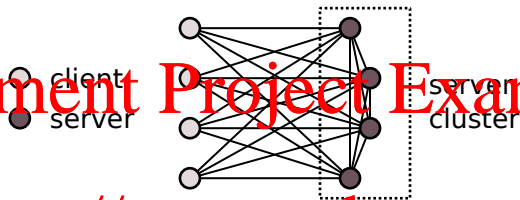
# Assignment Project Exam Help



The decentralized pattern became popular with the rise of peer-to-peer file sharing systems.

- Every peer process can directly communicate with every other peer process.
- Harder to implement than a centralized pattern – processes need to be both clients and servers.
- Data is distributed evenly over the peers.
- The failure of any peer is no worse than the failure of any other peer – the distributed system is very robust to process failure.
- Since peers are exposed to each other over the network their IP address becomes public – privacy/security.
- Resource capacity increases for the system as a whole for each new peer included in the system – scalability.

## Multi-server Pattern



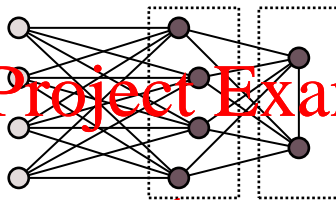
# Assignment Project Exam Help

<https://powcoder.com>

When a single server no longer supports the number of clients connecting to it, the multi-server pattern can be used to increase overall capacity.

- Each client can communicate with any of the server to obtain service. Clients never communicate directly with each other, similar to the centralized pattern.
- Considering only the multi-servers, they share many characteristics to the decentralized pattern: every server is like a peer to other servers – they can communicate directly with each other, harder to implement than a single server, data is distributed over the servers, the failure of a single server does not lead to system failure.
- The multi-server architecture exposes more server IPs to the clients – potential security issue.
- Servers can be geographically located at places that provide more uniform access for clients.

## Multi-tier Pattern



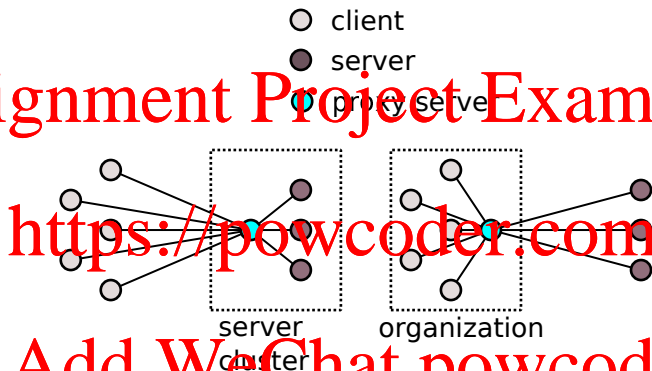
# Assignment Project Exam Help

<https://powcoder.com>

The multi-tier pattern is a horizontal view of distributed system functionality (as opposed to a vertical view given by layers).

- The tier furthest from the clients tends to be the data storage tier that maintains all of the data for the distributed system. Direct communication with clients never occurs which increases security.
- The tier closest to the clients typically never communicates among itself: this provides isolation and less overheads for synchronization. This tier accesses the data storage tier to obtain data for the clients.
- In a sense, each tier's processes are like client processes for the next tier inwards.
- More complicated than a multi-server and latency grows with the increase in the number of tiers.

## Proxy Pattern



A proxy is a process that sits between a client and a server, relaying request and responses.

- Clients never communicate directly with the server(s) – extra latency.
- All data flows through the proxy – bottleneck and single point of failure.
- A proxy used at the server side can relay requests to appropriate servers, e.g. a web proxy can relay to web servers and websocket servers.
- A proxy used at the organization side can manage client communication to the servers – security.

## Mobile code and mobile agents



# Assignment Project Exam Help

- In the mobile code pattern, a process provided by the server is executed on the client's machine, or *vice versa*. In a mobile agent pattern (not explicitly shown), data is copied along with the mobile code.
- For mobile code, offloading the work to the client reduces load on the server, or in the opposite direction, running client code on the server can avoid load at the client. Both cases avoid some amount of network communication. Mobile code is extremely effective as seen by Javascript in the browser as an example. It leads to "fat clients" (as opposed to thin clients such as the browser without any mobile code). Before Javascript became popular we used applets as mobile code.
- A mobile agent is mobile code that also maintains state that is copied with the agent's code.
- A mobile agent can move from one server to another, running tasks and collecting data. E.g. if the client requires to undertake a complex query, it may be better to express that as a mobile agent that is executed on the servers that have the data, since undertaking the query on the client may use excessive network resources.
- Executing code supplied by untrusted parties is a high security risk.
- Heterogeneous resources are a challenge.



## Process communication attributes

Edges in  $\mathcal{E}_n$  represent *required* communication between processes.

Application requirements on IPC may be described as process

communication edge attributes. For each  $(i, i') \in \mathcal{E}_n$ , let:

- $\omega_{i,i'}$  be the *bitrate* that is required by process  $i$  for communication with process  $i'$ , this may or may not equal  $\omega_{i',i}$  and communication requirements are usually assumed to be duplex,
- $\tau_{i,i'}$  be the maximum allowable *latency*, or the time it takes for the smallest transmittable amount of data, sent by process  $i$ , to be received by process  $i'$  which may or may not equal  $\tau_{i',i}$ ,
- $\rho_{i,i'}$  be the maximum allowable *packet loss*, which is the probability that a packet of data sent from process  $i$  to process  $i'$  is dropped or lost, which may or may not be equal to  $\rho_{i',i}$ .

If  $r = \text{network}$  is one of our resource types, we might write:

$$P_{i,\text{network}} = \sum_{(i,i') \in \mathcal{E}_n} \omega_{i,i'} + \sum_{(i',i) \in \mathcal{E}_n} \omega_{i',i}$$

## Machine communication attributes

Similarly, we can consider edges in say  $\mathcal{E}_m$  representing possible communication between machines, that is that two machines are reachable over the network. In many cases, we model the underlying network by describing the overall characteristics of the communication channels between all pairs of machines. For each  $(j, j') \in \mathcal{E}_m$  let:

- $\omega_{j,j'}^m$  be the *maximum bitrate* that can be achieved when sending data from machine  $j$  to machine  $j'$  which may or may not equal  $\omega_{j',j}^m$ , but in any case a duplex channel is usually assumed
- $\tau_{j,j'}^m$  be the *latency*, or the time it takes for the smallest transmittable amount of data, sent by machine  $j$ , to be received by machine  $j'$  which may or may not equal  $\tau_{j',j}^m$
- $\rho_{j,j'}^m$  be the *packet loss*, or probability that a packet of data sent from machine  $j$  to machine  $j'$  is dropped or lost, which may or may not be equal to  $\rho_{j',j}^m$ .

These three parameters are typically the most useful. We can see later how other aspects of the communication channel, such as *jitter* can be modelled as time varying latency – basically the bitrate and latency can change for each packet that is sent, which causes jitter.

## Applying graph theory I

The set of edges over processes is effectively a graph  $G = (V, E)$ , with vertex set  $V = \mathcal{N}$  and edge set  $E = \mathcal{E}_n$  and we can therefore apply graph theory to define properties of our distributed system architecture. The topology of the distributed system is concerned with the graph properties of its architecture that remain constant as  $n \rightarrow \infty$ . E.g. the number of *outgoing* edges from process  $i$ :

$$d_i^+ = |\{a \mid (i, a) \in \mathcal{E}_n\}|$$

and similarly the number of *incoming* edges,

$$d_i^- = |\{a \mid (a, i) \in \mathcal{E}_n\}|.$$

## Applying graph theory II

While the sum  $d_i^+ + d_i^-$  is sometimes interesting, mostly we want to consider the number of unique processes that each process is connected to and so we are interested in:

$$d_i = |\{a \mid (a, i) \in \mathcal{E}_n \text{ or } (i, a) \in \mathcal{E}_n\}|,$$

which in graph theory is called the *degree* of the vertex or the *number of neighbours* of process  $i$ . We can also define  $d = \max\{d_i\}$  as the degree of the graph, or the maximum number of neighbouring processes that any process has in the distributed system.

We can go further and write  $d$  as a function of the number of processes,  $d(n)$ . Now we have a topological function that is very useful for understanding our distributed system at any *scale*, i.e. at any size or value of  $n$ . E.g. if  $d(n) = n - 1$  then there are some processes that communicate with every other process in the system, whereas e.g. if

## Applying graph theory III

$d(n) = \sqrt{n}$ , or  $d(n) = \log n$ , or say  $d(n) = 2$  then we have a very different kind of distributed system topology.

Similarly we can examine a range of interest graph theoretic properties, e.g. the *diameter* of the graph. If  $\Phi_{i,i'} = \langle i, \dots, i' \rangle$  is a *shortest path* through  $G$  from  $i$  to  $i'$ , with path length being the number of edges in the path, then the diameter of  $G$  is the maximum of the shortest path lengths between all pairs of processes:  $\max_{i,i' \in \mathcal{N}} |\Phi_{i,i'}|$ .

This kind of analysis provides a means of defining some important concepts such as *scalability* that we discuss shortly. It also allows us to explore a rich space of concepts that apply to distributed systems and therefore help us develop distributed systems with well understood (read predictable) behaviour.

## Process and machine state

Each process maintains data which is the state of the process, and includes any data stored in the processes address space, let's say  $S_i^p$  is the state of process  $i$ .

- application data structures and objects,
- other data concerning the distributed system such as network addresses of other processes and IPC state such as socket objects and protocol state.

Each machine also maintains data which is the state of the machine, let's say  $S_j^m$  is the state of machine  $j$ , and in our model the state of the machine will include the state of the OS, and all of the permanent storage on the machine:

- data stored in files on behalf of processes,
- OS state including the state of processes running on the machine and network information specific to the machine.

Combined,  $S_i^p$  and  $S_j^m$  for all  $i$  and  $j$  represents the total state of the distributed system.

## Distributed system overheads

Processes are required to store information concerning the distributed system architecture, that is not information related to the actual application. Some well studied overheads related to the architectural model include

- If a process  $i$  is going to act as a client and communicate with say  $d_i^+$  other processes over the lifetime of the distributed system then presumably the process requires to know the network addresses of up to  $d_i^+$  other processes. We sometimes call this the *view* of the process or the *neighbour table*. The table contains  $d_i^+$  entries, each entry is constant size with respect to  $n$ .
- If a process  $i$  is going to act as a server and allow say  $d_i^-$  connections from other processes over the lifetime of the distributed system then presumably the process requires to store up to  $d_i^-$  connection objects, where each object is a constant size with respect to  $n$ . This is often combined into the same neighbour table state as above, as a single measure.
- Sometimes we are only interested in the neighbour table induced by considering undirected edges,  $d_i$ .
- If a process  $i$  can never communicate directly with process  $i'$  over the lifetime of the distributed system then presumably for information to be exchanged between process  $i$  and process  $i'$  that information will need to pass through at least  $|\Phi_{i,i'}| - 1$  intermediate processes.

# Scalability I

Earlier we loosely defined scalability as the ability for a system to increase its capacity linearly with an increase in resources. This would be trivial to achieve if the distributed system did not have overheads. Our topological model defined earlier allows us to consider properties of the distributed system as a function of  $n$  and  $m$ , i.e. as the number of processes and the number of machines grows larger. We also defined overheads of the distributed system in terms of its size. When the number of processes and/or the number of machines grow, the overheads also grow. If the overheads grow too large, too quickly, then an increase in the number of resources will not lead to a linear increase in the capacity.

- Consider a communication pattern where every process is required to communicate a message with every other process. For  $n$  processes that means  $n^2$  messages. As we double the size of the distributed system we need to grow the capacity of the network geometrically. This growth of network capacity is not scalable. Peer-to-peer systems avoid this by using communication patterns with a lower complexity, e.g.  $n \log n$ .



## Scalability II

# Assignment Project Exam Help

- We defined the neighbour table as the (maximum) amount of state that a process  $i$  must store over the lifetime of the distributed system,  $d_i^+$ . Consider  $d(n)$  (i.e. the maximum neighbour table size over all processes in the distributed system) for each of the patterns shown earlier and ask yourself how much state must a process in the distributed system store as  $n$  becomes very large. Generally if a process has to store state concerning every other process in the distributed system we say that its not scalable. We would prefer e.g. the overhead to be logarithmic to the size of the system, e.g.  $d(n) = \log n$ .
- Similar considerations apply to properties like the diameter of the architectural pattern.

## Adding time to the model

The model so far is *static* or *time-independent*. We can allow all of the model parameters to vary with time  $t$ , making the model *dynamic* or *time-dependent*.

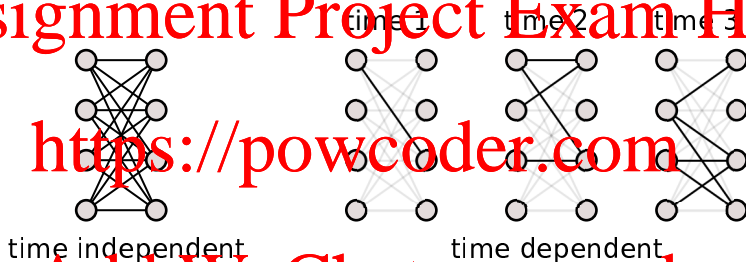
- $n(t)$ ,  $\mathcal{N}(t)$ , – the number of and ids of processes in the distributed system at time  $t$
- $m(t)$ ,  $\mathcal{M}(t)$ , – the number of and ids of machines in the distributed system at time  $t$
- $P_i(t)$ ,  $P_{i,r}(t)$  – the location of process  $i \in \mathcal{N}(t)$  and its requirement for resource  $r$  at time  $t$
- $C_j(t)$ ,  $C_{j,r}(t)$  – the resource  $r$  capacity for machine  $j \in \mathcal{M}(t)$  at time  $t$
- $\mathcal{E}_n(t)$ ,  $\mathcal{E}_m(t)$  – the IPC that occurs at time  $t$  and the machine connections and network characteristics at time  $t$

Some aspects may remain constant such as the set of all resources  $\mathcal{R}$  and other aspects are computed from the parameters, such as resource load  $L_{j,r}$ .

## Time dependent properties

## Assignment Project Exam Help

<https://powcoder.com>



Add WeChat powcoder

For example  $\mathcal{E}_n(t)$  may vary over time as above. We can now analyse properties of the distributed system with respect to a given point in time, or over a time interval.

## Discrete time

It is sometimes very convenient to model time as progressing in *discrete steps*. In this case we assume that  $t$  takes values  $t_0, t_1, t_2, \dots$  and that

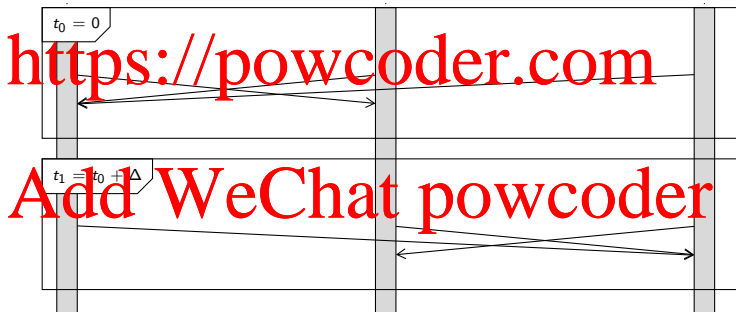
$t_{i+1} - t_i \in \Delta$ , where  $\Delta$  is some constant, typically  $\Delta = 1$ . When considering discrete time it is usually also assumed that all machines, and therefore all processes, progress *synchronously* in time, as if all of the machines are governed by an ideal global clock. This is a synchronous model. When using a synchronous model we tend to make assumptions like:

- a message sent at a given time  $t$  is received by time  $t + \Delta$ , i.e. in one time step
- all messages are constant size
- in each time step a machine can send at most  $k$  messages and receive at most  $k$  messages (i.e.  $2k$  messages in total per time step)

The synchronous model is useful for analysing distributed algorithms in a formal sense, where the complexity of the distributed algorithm (number of time steps to complete) is expressed in terms of communication complexity (required information that must be communicated as part of the algorithm).

Let's assume that each process can send 1 message and receive up to 2 messages in each round. Send and receive times are synchronized.

# Assignment Project Exam Help



## Continuous time

# Assignment Project Exam Help

We may wish to consider time progressing *continuously* or *asynchronously* where  $t$  is a real number. We may still consider points in time  $t_0, t_1, t_2, \dots$  however all that we assume is that  $t_a > t_{a-1}$  for all  $a > 0$ . When using an asynchronous model we tend to make assumptions like:

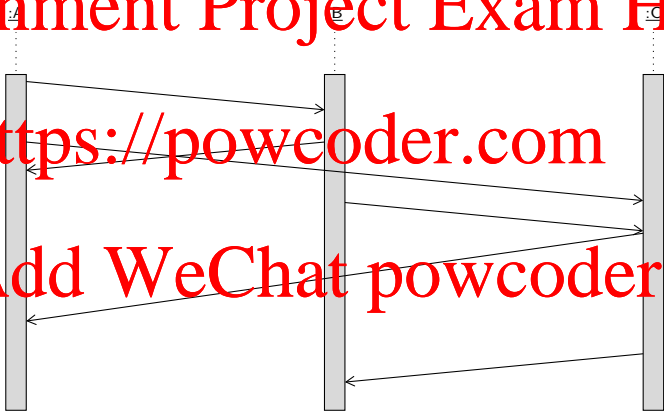
- a message sent at a given time  $t$  is received at some time  $t' > t$
- the time taken to send a message is at least as long as it takes for the sending machine to transmit the message, which is a function of the size of the message and the available outgoing bitrate of the machine, or similarly for the receiving machine to receive the message which ever is the longer time
- only one message can be being transmitted or received at any point in time

Processes send messages as needed and messages take some time to be received. Send and receive times are not synchronized.

# Assignment Project Exam Help

<https://powcoder.com>

Add WeChat powcoder



## Defining failure

All distributed systems are subject to a real-world phenomenon called *failure* which arises from:

- *faults* – unintended and usually unknown defects in hardware manufacturing and/or software programming,
- *exceeding specified operating conditions* – random environmental conditions, such as mechanical shock, electromagnetic interference, exceeding capacity such as running out of storage space, or user behaviour, that exceeds the range specified for “failure-free” operation, including operation of hardware beyond its specified lifetime.

A *Failure Model* states the types of failure that are being considered, i.e. what types of failure will the distributed system be subject to, and therefore exclude all remaining types of failure. Perhaps the remaining types of failure are assumed to be handled by middleware or the OS/hardware, or else they are ignored and the system's behaviour is undefined when such failures happen.



## Types of failure I

Failure manifests in a number of ways, listed here in a loose order of *severity*:

- *error indicators* – error codes returned from methods including returning null values, cases where a method call has failed, and error messages returned from other processes to indicate some kind of failure
  - can arise from incorrect parameter values passed to the method, requested machine resources being temporarily unavailable or out of capacity, network errors, corrupted data read from storage, and incorrect user input (e.g. specifying a location store a file that is not allowed by the OS),
  - processes have a chance to handle the error condition
  - if the process continues obliviously to the error condition then further, more severe failure may arise – the process is now in an *undefined state*,
- *exceptions* – may terminate threads if they are not handled, may terminate the process if the main thread terminates
  - if a method does not return error codes and the expected operation of the method can not be achieved then an exception may be raised
  - users may interrupt the process (user is signalling an error condition)
  - out of memory, out of disk space, division by zero – the execution of code can't continue, returning is not an option
  - threads can catch exceptions and handle them, if not the thread may be terminated and this may terminate the process – if a thread is terminated or “dies” without being handled then the process may continue to fail

## Types of failure II

- *process termination* – OS or user terminates the process
  - the process may be attempting to access memory or other machine resources in an invalid way and/or in way that is potentially harmful to the OS and the machine
  - the user wants to immediately prevent the process from executing
  - the OS may be trying to avoid OS failure, e.g. when swap space is exhausted then the OS may terminate the process that is using the largest amount of memory
  - processes usually cannot catch or handle this kind of failure, the process terminates without any notice
- *OS/hardware failure* – the OS and/or machine resources fail
  - OS failure can lead to failure of all processes on the machine – restarting the machine may be required
  - storage devices fail entirely – none of the data is available
  - network devices fail entirely – the network is not usable
  - the machine fails – may or may not be able to be restarted – machine state including permanent storage may or may not be available
- *power supply failure* – many machines connected to the same power supply may fail at the same time
  - this is an example of correlated failure
  - individual machines may or may not lose permanent storage
- *network failure* – computer network equipment can fail as well
  - packets are dropped – largely due to queueing becoming full under high load,

## Types of failure III

# Assignment Project Exam Help

- physical connections are subject to interference (such as wireless connections) and/or break down and need to be repaired
- network configuration errors – routing fails to operate correctly
- networks become separated (partitioned) where machines on one network cannot communicate with machines on another network

<https://powcoder.com>

Add WeChat powcoder

## Failure detection

- Failure may be reported to the process with error codes and exceptions. However not all types of failure are reported, many types of failure need to be *detected*.

*Silent failure* – there is no failure reporting mechanism such as error codes or exception, processes must explicitly detect such failure:

- dropped packets – the network does not report that a packet was dropped
- processes hanging – a “hung” process becomes unresponsive, it can happen if the process enters an infinite loop, blocks on a condition that never arises, deadlock, or if a thread dies e.g. an IO thread or a thread handling a request
- distributed failure – deadlock can occur between processes in a distributed system
- OS failure – thrashing can lead to all processes on the machine becoming unresponsive
- device failure – can cause the OS to block indefinitely, which in turn can cause processes to block indefinitely
- Failure of a remote process* in a distributed system is not automatically reported to other processes in the distributed system – it must be detected
  - unless there is another distributed system process on the same machine as the remote process, it is technically impossible to be able to tell the difference between some kinds of network failure and process failure – no response to communication requests could be either
  - sometimes communication requests are explicitly “refused” by the remote machine, which is indicative of remote process failure
  - if a remote process cannot be communicated with for some time, the distributed system has no alternative but to label that as failure – being unresponsive is considered failure

## Failure handling I

When a failure has been detected then it should be handled. If distributed system can continue operating correctly and at full capacity under the presence of failures we say that it is *fault tolerant*. This should not be confused with the notion of ‘tolerating’ faults as discussed below.

- *fail-stop* – the failure (termination or otherwise) of a process can be detected certainly by other processes in the distributed system
  - Failure is not always detected. A process may fail and continue to operate without that failure being known. A process may become unresponsive and appear to have failed, when it has not.
  - Without the fail-stop assumption, when a process has failed or has been deemed by the remaining processes to have failed (e.g. if the process is unresponsive), the process is deliberately terminated to avoid any further potential for errors. Other processes will consider that processes terminated, and even if it “wakes up” later and starts responding, it will be told that it should terminate since it is no longer considered part of the distributed system.
- *graceful failure* – the distributed system can continue to operate, probably with degraded performance, in the presence of faults. A distributed system with enough reliability can continue to operate even though some machines may have failed, but the system may not provide the same response times that it did before the failure. Another example is that packet loss may cause only a minor degradation in video stream quality.

## Failure handling II

- *tolerating faults* – in this case the users of the system are required to tolerate a certain amount of failure: “can’t provide service, please try again later”. Most users of commercial operating systems tolerate failure in the OS: they restart their computer. They don’t take it back to the store and say it crashed and doesn’t work and they want another one. Again with the example of video streaming, most users will accept the occasional degradation in video quality when faults occur like packet loss.
- *failure masking* – the distributed system attempts to hide the failure by e.g. retrying the operation transparently. This is a transparency challenge. Some failures cannot be masked since e.g. if the network is down then the streaming video can’t be delivered and the user will notice this (we could try to play pre-cached commercials in the hope that the network will come back up soon to distract the user from noticing the fault).
- *failure recovery* – in some cases, failure needs to be recovered from, because the state of the distributed system has been affected by the failure, e.g. data has been lost, or else the state of the system is no longer valid.

## Failure handling III

# Assignment Project Exam Help

- If the system has redundancy, e.g. using multiple copies or erasure codes to store state, or having multiple processes on independent machines that are undertaking the same computations, then recovering from failure means make use of the redundancy, but as well, re-establishing the same level of redundancy in case future failures occur. Failure reduces the amount of redundancy, and so it must be recovered for the system to be at full operating capacity.
- Check-pointing system state, i.e. storing the state to permanent storage on a regular basis is a technique that provides for failure recovery. If state is lost, e.g. if a process fails, then the process can be restarted using the last check-point, which may involve the process having to redo some calculations again in order to “catch up” with other processes in the distributed system. Distributed check-pointing is a well studied problem in distributed systems. Complex check-pointing will check-point state at different levels: e.g. process state and machine storage state. In case of machine storage failure, we need to keep redundant copies over multiple machines.

## Modelling and analysing failure I

The exact time and type of the next failure to occur in a specific system is difficult to accurately predict, for if it were easy to predict then engineers would likely already have removed its possibility. Therefore it is widely accepted to model failure by estimating the *mean time to failure* (MTTF) which gives rise to a failure rate:

$$\lambda = \frac{1}{\text{MTTF}}$$

The MTTF is a measure of *reliability*. Higher MTTF means greater reliability.

We usually assume that the probability of the system failing within the next  $t$  seconds is given by the cumulative distribution function of the Exponential distribution:

$$\mathbb{P}[\text{failure} < t] = 1 - e^{-\lambda t}$$



## Modelling and analysing failure II

The Exponential distribution,  $f(t; \lambda) = \lambda e^{-\lambda t}$ , assumes that we have no information about when the next failure will occur.

If we are using a discrete-time model, we may want to estimate the number of failures,  $x$ , that occurred in the time interval  $\Delta$ , which is given equivalently by use of the Poisson distribution:

$$\mathbb{P}[\text{failures} = x; \lambda, \Delta] = \frac{(\lambda \Delta)^x e^{-\lambda \Delta}}{x!}$$

Consider  $n$  processes, where each process can fail independently at random according to our model above. The probability that none of the processes fail within the next  $t$  seconds is:

$$(1 - \mathbb{P}[\text{failure} < t])^n = (e^{-\lambda t})^n = e^{-n \lambda t}$$

## Modelling and analysing failure III

The probability that at least one of the processes fails within the next  $t$  seconds is therefore:

Assignment Project Exam Help

For example, if MTTF for each process is 1 week, i.e. each process fails independently at randomly on average after 1 week of operation, then for a distributed system with 100 processes, the probability that at least one of the processes has failed in the first day of operation is:

$1 - e^{-\frac{10}{7}} \approx 0.76$

Add WeChat powcoder

If we had 100 processes in our distributed system then the probability of at least one process failing in the first day becomes practically certain. Put another way, with 100 processes, the effective failure rate is now  $100\lambda$  which gives an effective MTTF of  $\frac{7}{100}$  days or 1.68 hours: the system will fail on average after 1.68 hours or about 100 minutes of operation.

## Modelling and analysing failure IV

# Assignment Project Exam Help

As the number of components in a system increase, the system's chance of failing increases. Thus, for large-scale distributed systems we need to invest in more fault tolerance techniques in order for the system to operate for sufficient periods of time to be useful.

<https://powcoder.com>  
Add WeChat powcoder

## Availability Model

While we are talking in terms of probability, if we let  $X = 1$  if the distributed system is *available for use* when we try to use it and  $X = 0$  otherwise, and we model  $X$  as a random variable, then the *availability* can be defined as:

$$\mathbb{P}[X = 1]$$

i.e. the probability that the distributed system is available for use when we try to use it. If the availability is 0 then the system is never available, and if the availability is 1 then it is always available. The availability can also be seen as the fraction of total time for which the system is available, e.g. 6.9 days of every 7 days the system is available and 0.1 days it is not. Thus  $\mathbb{P}[X = 1] \Rightarrow 6.9/7 \approx 0.99$ .

Of course, this is just a model. We may know that the system is down regularly on Mondays at 1am, while our model assumes that downtime could happen at any time. We can always refine the model to take additional knowledge into account.

## Goals, threats and mechanisms

- Security of a distributed system is achieved by securing processes, communication channels and protecting objects they encapsulate against unauthorized access.
  - *Access rights* specify who is allowed to perform operations on data in the distributed system.
  - Each operation requested in the distributed system is associated with a *principal* – the user or agent authorized to undertake such an operation.
- Possible threats from an enemy.
  - Threats to processes: Servers and clients cannot be sure about the source of the message. Source address can be faked.
  - Threats to communication channels: Enemy can copy, alter or inject messages.
  - Denial of service attacks: overloading the server or otherwise triggering excessive delays to the service.
  - Mobile code: performs operations that corrupt the server or service in an arbitrary way.
- Addressing security threats:
  - Cryptography and shared secrets: encryption is the process of scrambling messages.
  - Authentication: providing identities of users.
  - Secure Channel: Encryption and authentication are used to build secure channels as a service layer on top of an existing communication channel. A secure channel is a communication channel connecting a pair of processes on behalf of its principles.