**1. what is the reason for distribution transparency in distributed systems?**

The goal is to hide the fact that processes are physically distributed across multiple computers. A distributed system that is able to present itself to the user and applications as if it were only one computer system is called transparent.

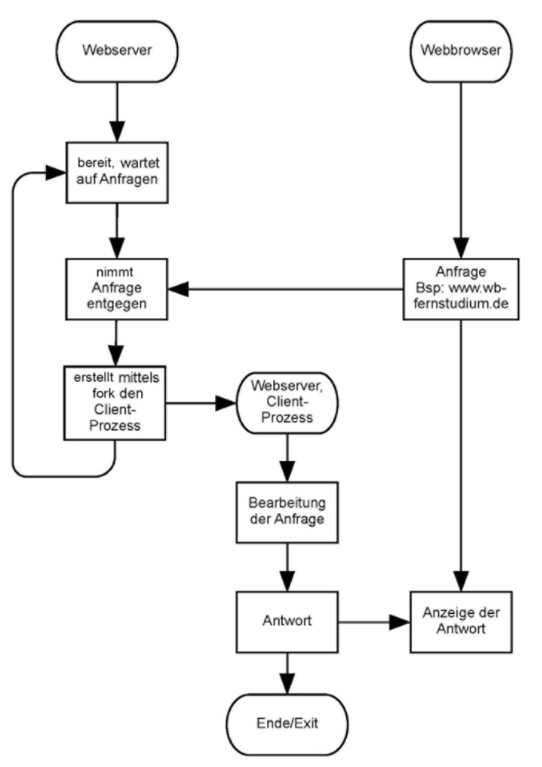
There are several types of transparency:

|  |  |
| --- | --- |
| Access | Hides differences in data representation and the way a resource is accessed |
| Location | Hides the place where a resource is located |
| Migration | Hides so that a resource can be moved to another location |
| Relocation | Hides so that a resource can be moved to another location while using it |
| Replication | Hides that a resource is replicated |
| Concurrency | Hides the fact that the resource is used by several competing users at the same time |
| Error | Hides the failure and recovery of a resource |
| Persistence | Regardless of whether a particular component is active or inactive, the system shows the same behavior externally, through appropriate activation or deactivation mechanisms. |
| Transaction | Hides the mechanisms that provide ACID properties in DB components of the application. |

**2. why is marshalling/unmarshalling necessary in distributed object-based systems?**

Marshalling and unmarshalling is necessary in distributed object-based systems because it enables communication between different parts of the system. Marshalling is the process of converting data into a standardized format that can be transmitted over a network. Unmarshalling is then the process of converting that data back to its original form. This is important for distributed systems because it allows different parts of the system to communicate with each other without having to understand each other's native formats. This process allows data to be easily exchanged between different parts of the system.

**Fig. 3.4 shows a parallelization of requests by means of a fork. What would the process look like if the parallelization was implemented using threads?**



If request parallelization were implemented with threads instead of forks, the process would look something like this:

- Create a new thread for each request

- Newly created thread can run independently

- Web server can immediately wait for requests again

- When one thread is finished it sends the result to the browser that made the request.

I.e. the process is completely the same, only that instead of a fork always one thread per request is created.

**4. semaphores**

You have a constrained resource called a toilet and another constrained resource called a washroom. Four toilets exist and the washroom can hold six people. The toilets can only be entered and exited through the washroom. Using semaphores, model how to ensure that no jams can occur.

A semaphore is used for the restrooms and initialized with value 4. If a person wants to use the toilet, they must get "permission" from the semaphore\_T. If a person goes into the toilet the value is reduced by 1. If a person leaves the toilet the value is increased by 1 again. If the value is 0 all toilets are occupied.

However, to get into the toilets, people have to pass through the washroom. This is also controlled by a semaphore\_W. This is initialized with the value 6. The problem is now solved so that there are four ways of using it:

1. directly into the toilet: possible if Semaphore\_T > 0. Here also Semaphore\_W must be fetched and thus Semaphore\_W > 0. Action: decrement Semaphore\_T and decrement Semaphore\_W.

2. only to the washroom: possible if Semaphore\_T = 0. Here only Semaphore\_W must be fetched and be greater than 0. Action: decrement semaphore\_W.

3. from the washroom to the toilet: possible for persons who already have Semaphore\_W. Action : decrement semaphore\_T.

4. leaving the toilet: Possible for people who have semaphore\_T. Action: Increment semaphore\_T and increment semaphore\_W.

This leads now to the fact that if a person goes directly on the toilet it reserves itself directly also place in the washroom around this to be able to leave. There can always be at most so many persons that the value of the Semaphore\_W is large enough so that all in the toilets can also again in the washroom. If 4 people are in the toilet, at most 2 can still enter the washroom. This prevents the deadlock.

A person who is already in the washroom has already taken the Semaphore\_T, therefore he only has to decrement the Semaphore\_T at action 3 and has already reserved his place in the washroom. When the person finally leaves the washroom, the place in the washroom is released together with the place reservation for the washroom.