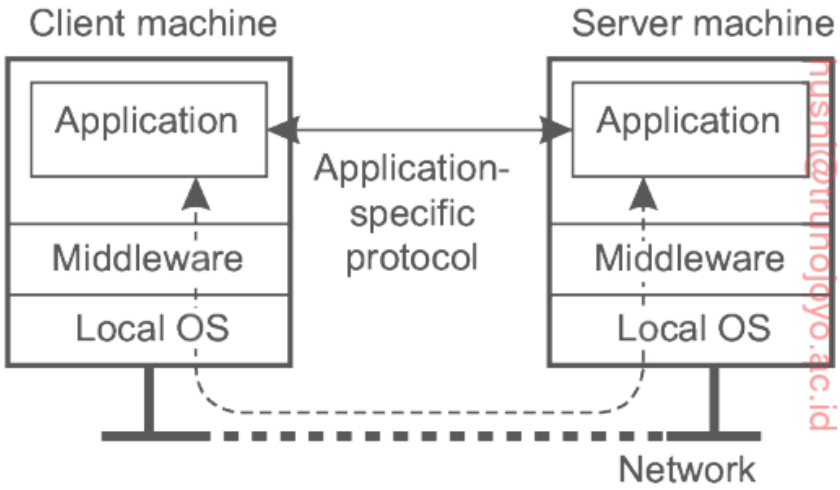
Clients

Networked user interfaces

A major task of client machines is to provide the means for users to interact with remote servers. Everything is processed and stored at the server.

2 ways in which this interaction can be supported:

1. For each remote service, the client machine will have seperate counterpart that can contact service over the network. Example: Calender running on a user's smartphone that needs to sync with a remote, possibly shared calender. In this case, fig.(a)



2. Providing a direct-access to remote service by offering only a convenient UI. Effectively, this means that the client machine is only used as a terminal with no need for local storage, leading to an application-neutral solution as in fig.(b). This is a **thin-client approach**.

Client-side software for distribution transparency

In many cases, parts of the processing and data level in a client-server application are executed on the client-side as well (client-side scripting). Special class formed by embedded software: automatic teller machines (ATMs), cash registers, barcode readers, TV set-top boxes, etc.

Ideally, a client should not be aware that it is communicating with remote processes. In contrast, distribution is less transparent to servers for reasons of performance and correctness.

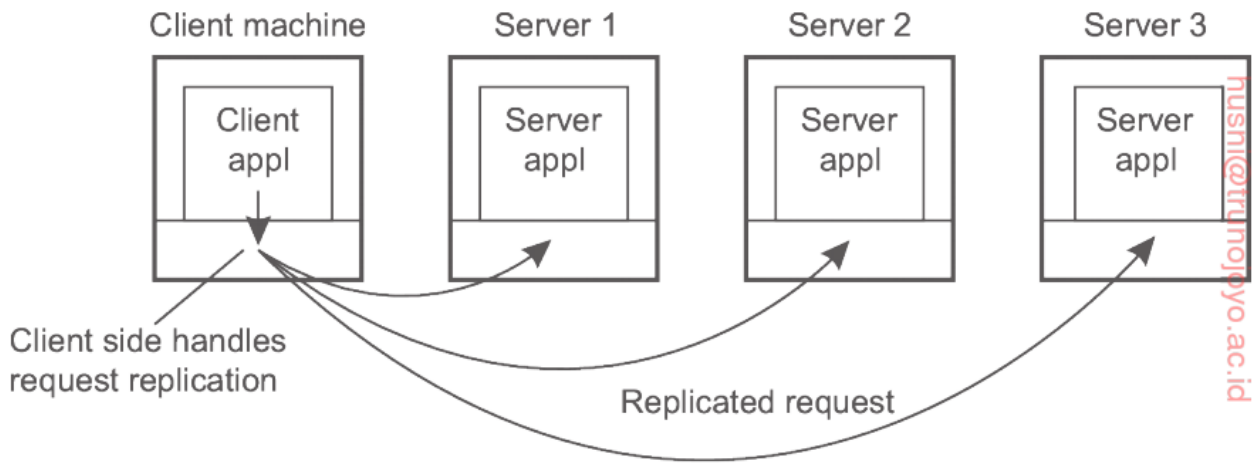
Access Transparency

Access transparency is generally handled through the generation of a **client stub** from an interface definition of what server has to offer. The stub provides the same interface as the one available at the server, but hides the possible differences in machine architectures, as well as the actual communication. The client stub transforms local calls to messages that are sent to the server, and *vice versa* transforms messages from the server to return values as one would expect when calling an actual procedure.

Client Stub => procedure call => message => server stub => message => return value => client

Replication Transparency

Many distributed systems implement **replication transparency** by means of client side solutions. For example, imagine a distributed system with replicated servers, such replication can be achieved by forwarding a request to each replica. Client-side software can transparently collect all responses and pass a single response to the client application.



Failure Transparency

Regarding **failure transparency**, masking communication failures with a server is typically done through **client middleware**(400s, 500s response statuses, I think!). For example: a client middleware can be configured to repeatedly attempt to connect to a server, or perhaps try another server after several attempts. There are even situations in which client middleware returns data it had cached from the previous sessions as in the case of web browsers.

Concurrency Transparency

Finally, **concurrency transparency** can be handled through special intermediate servers, notably transaction monitors, and requires less support from client software.