Middlewares

Goal: Achieving Openness.

Design Types

- Wrappers
- Interceptors

Wrappers

- Also called **Adapter**.
- Is a special component that offers an interface acceptable to a client application.
- Functions are transformed into those available at the component.
  Meaning if we have an integer and the component has a function that accepts only a float, then the interface transformed into a float
- Solves the problem of **incompatible interfaces**.
- Example: an **object adaptor** is a component that allows applications to invoke remote objects although those objects may have been implemented as a combination of library functions.
- Developing adaptors specific to an application would require $\mathcal{O}(N^2)$ wrappers. (Every application will have $N-1$ wrappers to communicate with $N-1$ applications.)

Reducing the number of wrapper through middleware

BROKERS

Broker is a logically centralised component that handles all access between different applications.

Often used type of broker is the **message broker**, to which applications simply send requests containing information on what they need. The Broker having knowledge of all relevant applications, contacts the appropiate applications, possibly combines and transforms the reponses and returns the result to the intial application.

We need atmost $2N = \mathcal{O}(N)$ wrappers, because we need two links for request and response.

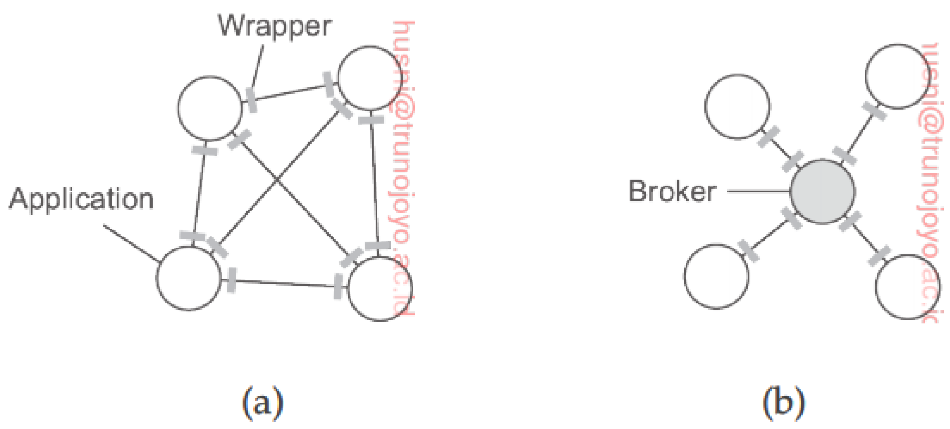

Figure 2.13: (a) Requiring each application to have a wrapper for each other application. (b) Reducing the number of wrappers by making use of a broker.

Interceptors

- **Interceptor** is software construct that will break the usual flow of control and allow other (application specific) code to executed.
- Primary means for adapting middleware to the specific needs of an application.
- Plays an important role in making middleware open.
- In many cases, having limited interception facilities will improve the management of the sodtware and distributed system as a whole

Basic idea

An object A can call a method that belongs to object B, while the latter resides on a different machine than A.

Three steps for remote object invocation:

1. Object A is offered a local interface that is identical to the interface offered by object B. A calls the method available in that interface.
2. The call by A is transformed into generic object invocation, made possible by a through a general object-invocation interface offered by the middleware at the machine where A resides.
3. Finally, the generic object invocation is transformed into a message that is sent through the transport-level network interface offered by A's local operating system.

`callbyA()` -> Generic object -> message -> send to B

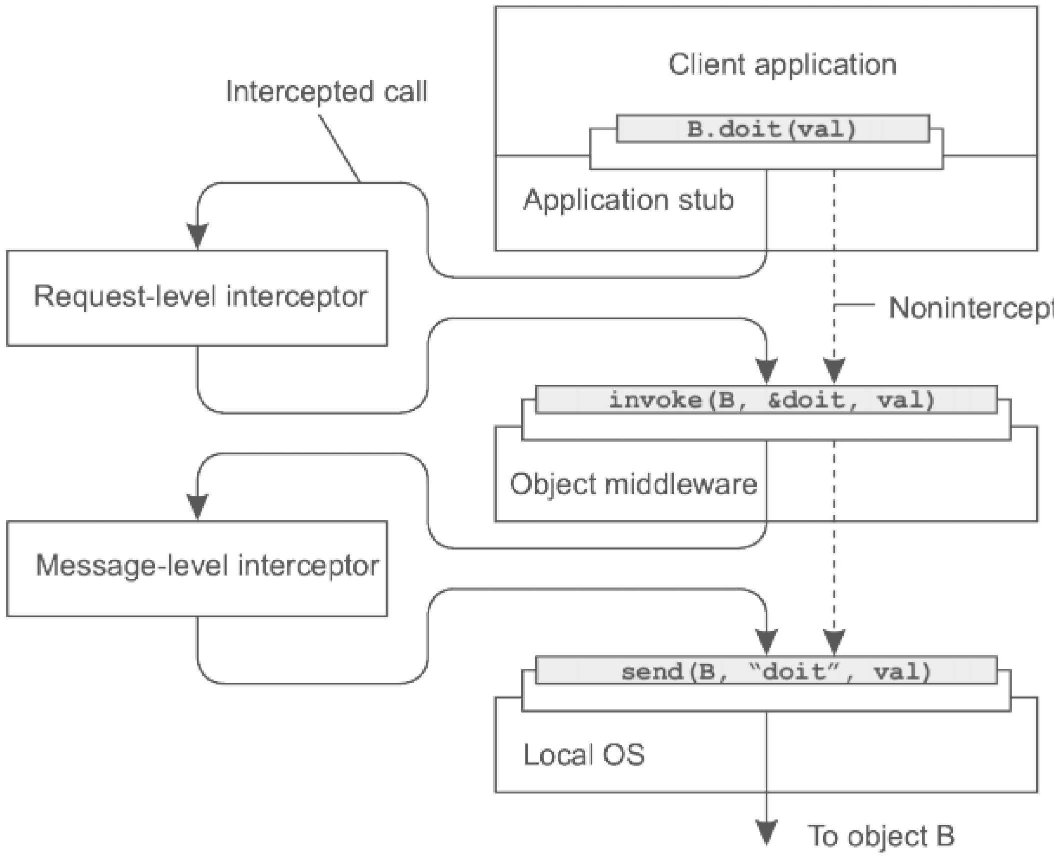

Figure 2.14: Using interceptors to handle remote-object invocations.

EXPLANATION

- The call `B.doit(val)` is transformed into a generic call `invoke(B, &doit, val)` with a reference to B's method and the parameters that go along with it.
- Imagine B is replicated, then each replica should actually be invoked, **request-level interceptor** simply calls `invoke(B, &doit, val)` for each replica.
- The **message-level interceptor** assists in transferring the invocation to the target object.

Modifiable Middleware

What *wrappers* and *interceptors* offer are means to adapt and extend the middleware. The environment in which distributed applications are executed changes continously.

Changes include:

- Mobility
- Strong variance in *quality-of-service* of networks
- failing hardware
- battery drainage, among others.

All tasks which are the consequence of the reactions resulting from these changes are placed in the middleware. Changes can't be done by temperarily shutting down the distributed application. Changes should be made *on-the-fly*.

Middleware may need not only be adapted but that we should be able to purposely modify it without bringing it down.

Example: Replacing software components at runtime.

Most popular approach of modifiable middleware: **Dynamically constructing middleware from components**.

Dynamic nature supports late binding, **where modules can be loaded and unloaded at will**.

State management between calls to a component requires special measures.