Patrones Arquitecturales

Conceptos Clave

- Monolítico: todo está en un solo lugar.

- Asincrónico: las cosas se mandan a que se procesen.. Y en algún momento volverán.
  - Lo “opuesto” es Sincrónico: las cosas se esperan hasta que se terminen.

- Distribuidos: computadoras conectadas para resolver problemas juntas. Son los clúster o grids. Existen muchos modelos.

Aparte del N-Capas, existen más.

- Arquitectura por eventos
- Microkernel
- Microservicios
- Basada en Espacio

Arquitectura por eventos

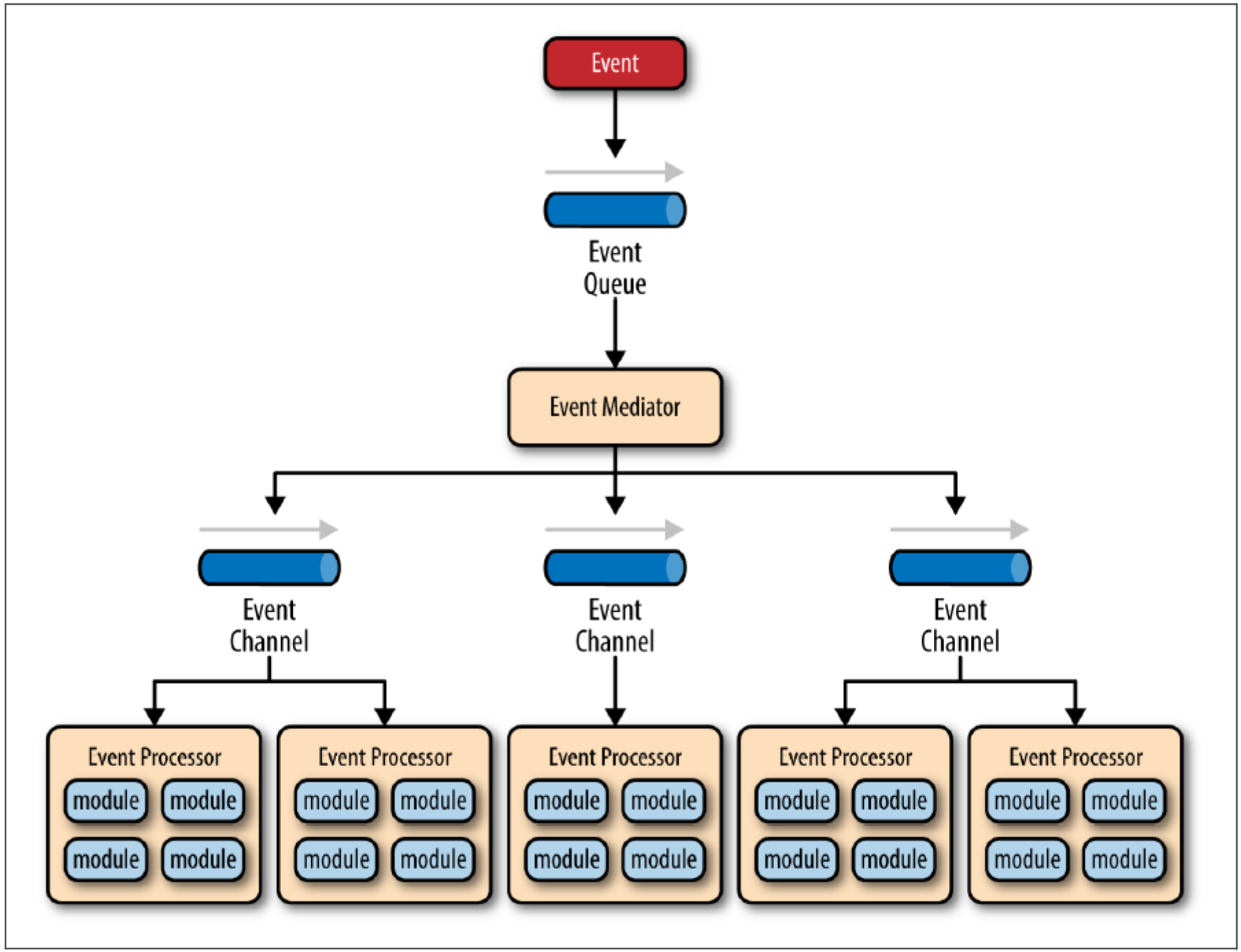
- Es un patrón común en sistemas distribuidos y asincrónicos, porque permite gran escalabilidad.
- Muy desacoplada.
- Procesa un evento a la vez, de forma asincrónica.
- Hay 2 sabores: con Mediador o con “Broker” (negociador?)

Arquitectura por eventos

- Cuando hay un mediador, hay un único objeto que actúa como director de orquesta.

- Hay 2 tipos de eventos: iniciador o de procesamiento.

- Hay 4 tipos de objetos:
  - La cola de eventos
  - El mediador propiamente dicho
  - Los canales
  - Los procesadores

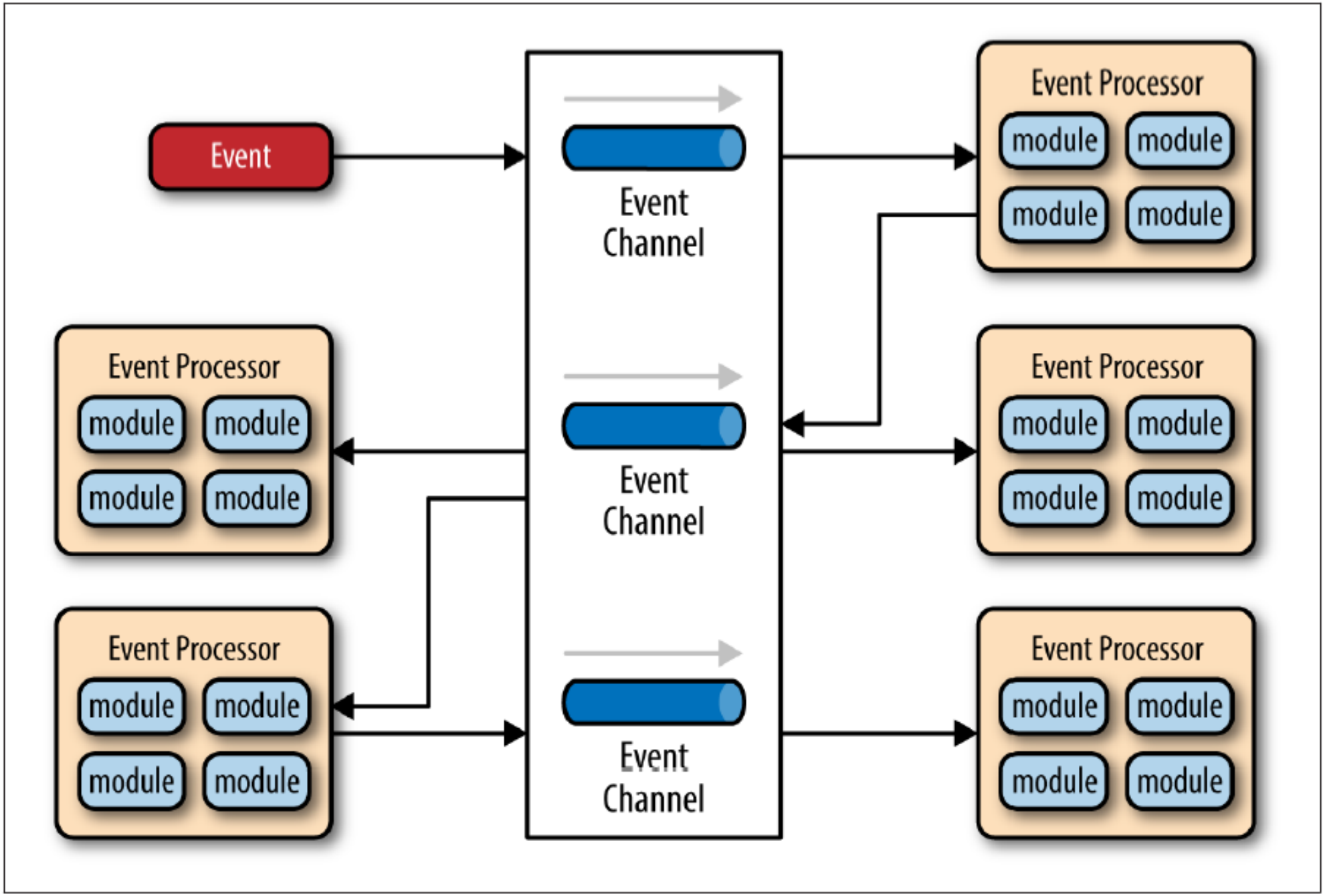

Event
Event Queue
Event Mediator
Event Channel
Event Channel
Event Channel
Event Processor
module
module
module
module
Event Processor
module
module
module
module
Event Processor
module
module
module
module
Event Processor
module
module
module
module
Event Processor
module
module
module
module

Arquitectura por Eventos

- Cuando se usa un bróker NO hay mediador único: el mensaje (evento) se distribuye a toda la red y alguien lo toma.

- Es muy útil cuando no hay flujo de procesamiento de eventos → no se necesita alguien que dirija.

- Hay 2 componentes:
  - El bróker
  - El procesador


Event
Event Channel
Event Channel
Event Channel
Event Processor
module
module
module
module
Event Processor
module
module
module
module
Event Processor
module
module
module
module
Event Processor
module
module
module
module
Event Processor
module
module
module
module

Arquitectura de eventos

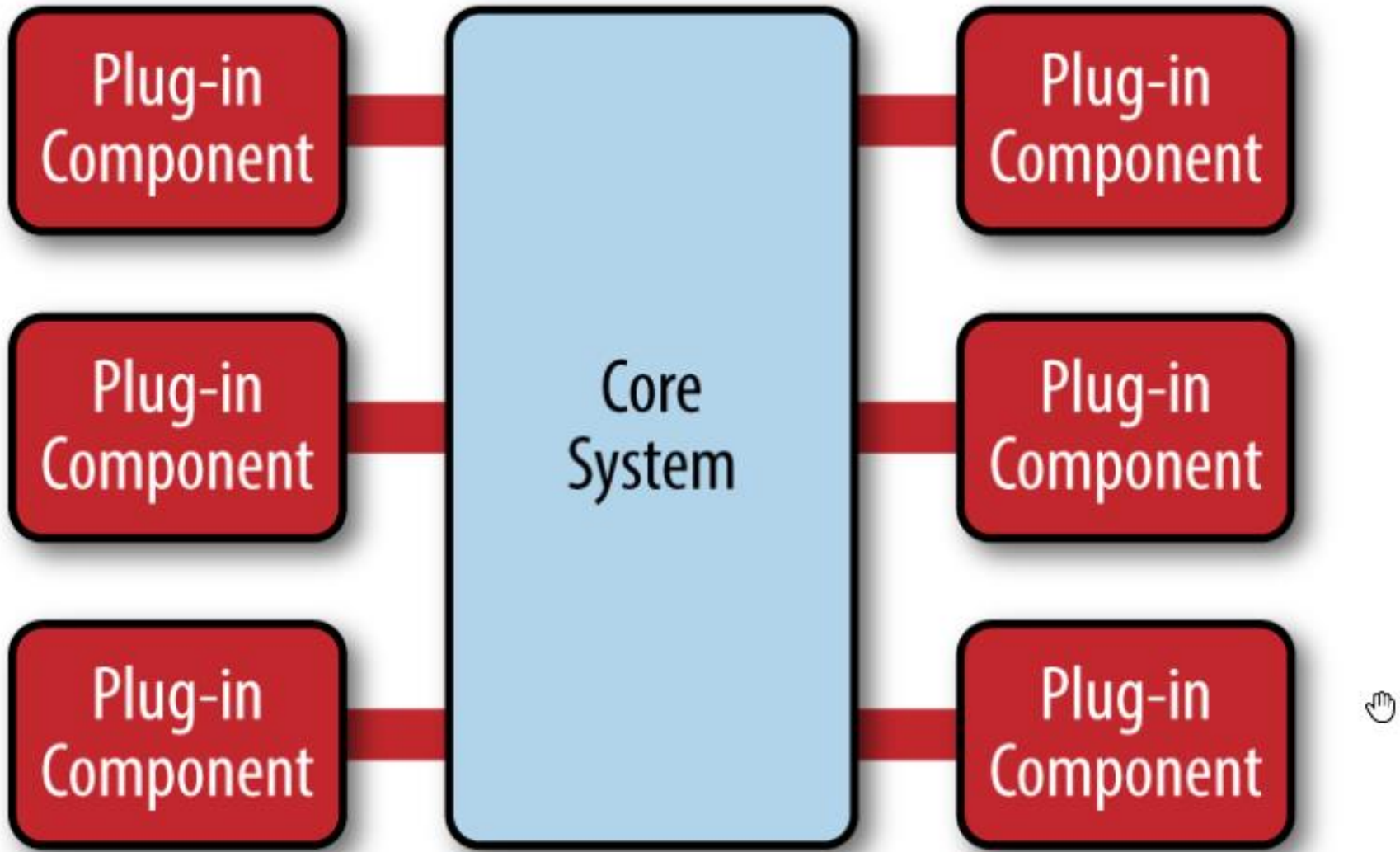
- Tiene problemas de manejo de transacciones, especialmente cuando hay orquestación.
- Los procesos deben ser muy granulares.
- El mantenimiento de las interfaces es CRÍTICO.
- Es muy fácil responder a cambios requeridos en componentes aislados.

Arquitectura de Eventos

- Es fácil hacer deploy
- Dificil de probar
- Permite alcanzar tasas de rendimiento elevadas (no en vano las supercomputadoras son casi siempre sistemas distribuidos)
- Muy escalable.
- Difícil de programar.

Arquitecturas de Microkernel

- La idea es tener un sistema principal (CORE) y módulos separados (plug-ins)
- La lógica está distribuida de forma independiente entre el core y los plugins.
- Es muy común en sistemas operativos.


Plug-in Component
Plug-in Component
Plug-in Component
Core System
Plug-in Component
Plug-in Component
Plug-in Component

Arquitectura de Microkernel

- Los plugins son entidades individuales. Auténticos componentes.
- El core necesita saber como conseguir plugins y como conectarse a ellos, amén de como activarlos.
- Es muy útil en sistemas “embebidos”. Y es una forma común de organizar microservicios.
- Las interfaces y su mantenimiento son críticas.
- Es fácil implementar cambios independientemente.

Arquitectura de Microkernel

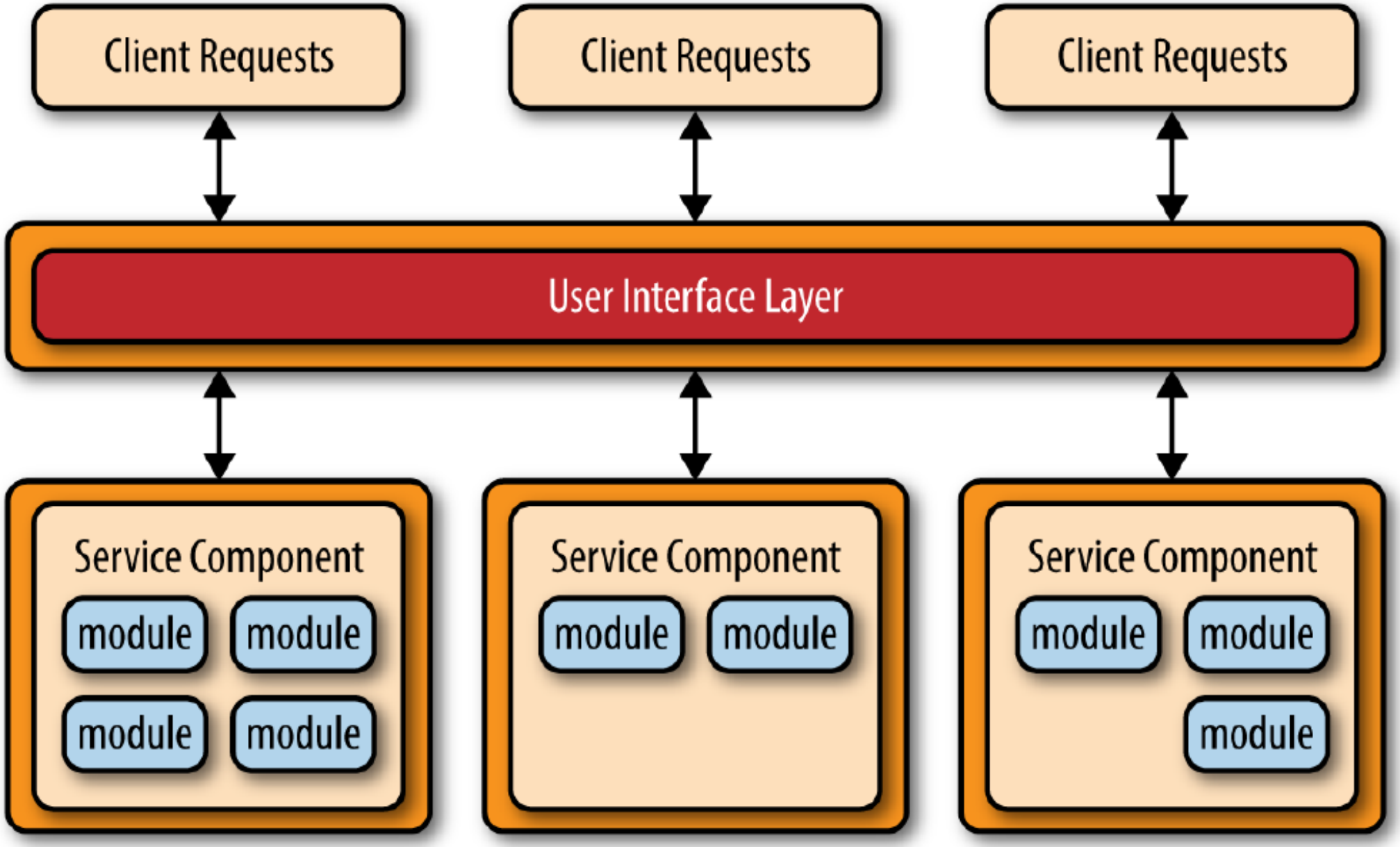
- Es fácil desplegar el core y los plugins dependientes.
- Es fácil probar
- No es muy escalable. EL kernel es prácticamente un monolito.
- Es difícil de progamar. Requiere mucho control de contratos y diseño del kernel.

Arquitectura de Microservicios

- La moda. Hay mucha confusión por su “hype”. No hay forma clara de hacerlos. Cada tecnología tiene su sabor.

- La idea general es tener desplegadas pequeños componentes llamados SERVICE COMPONENTS. La granularidad requerida es muy alta.

- Cada Service Component tiene una función única o una porción de procesamiento específica.

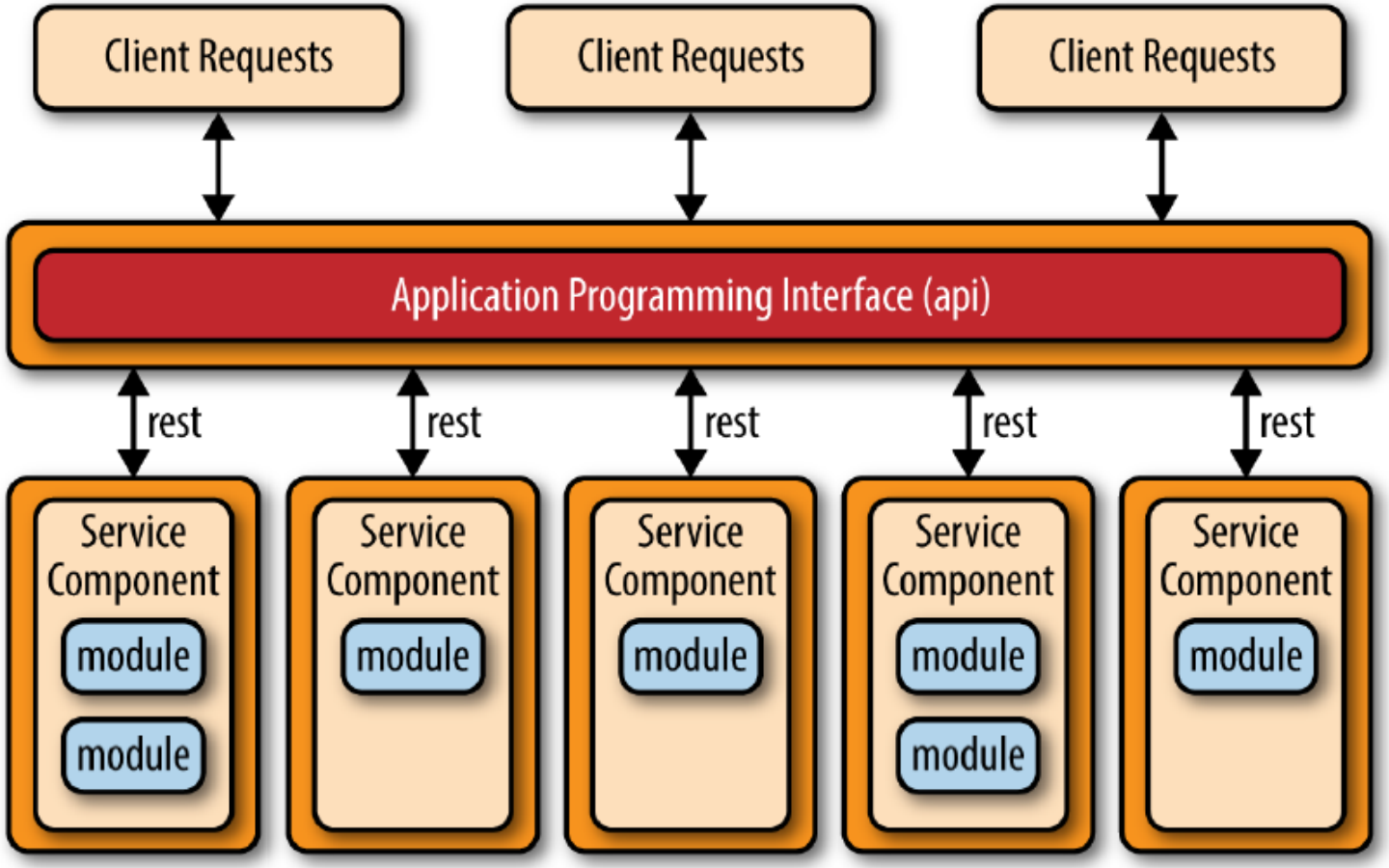

Client Requests
Client Requests
Client Requests
User Interface Layer
Service Component
module
module
module
module
Service Component
module
module
Service Component
module
module
module

Arquitectura de microservicios

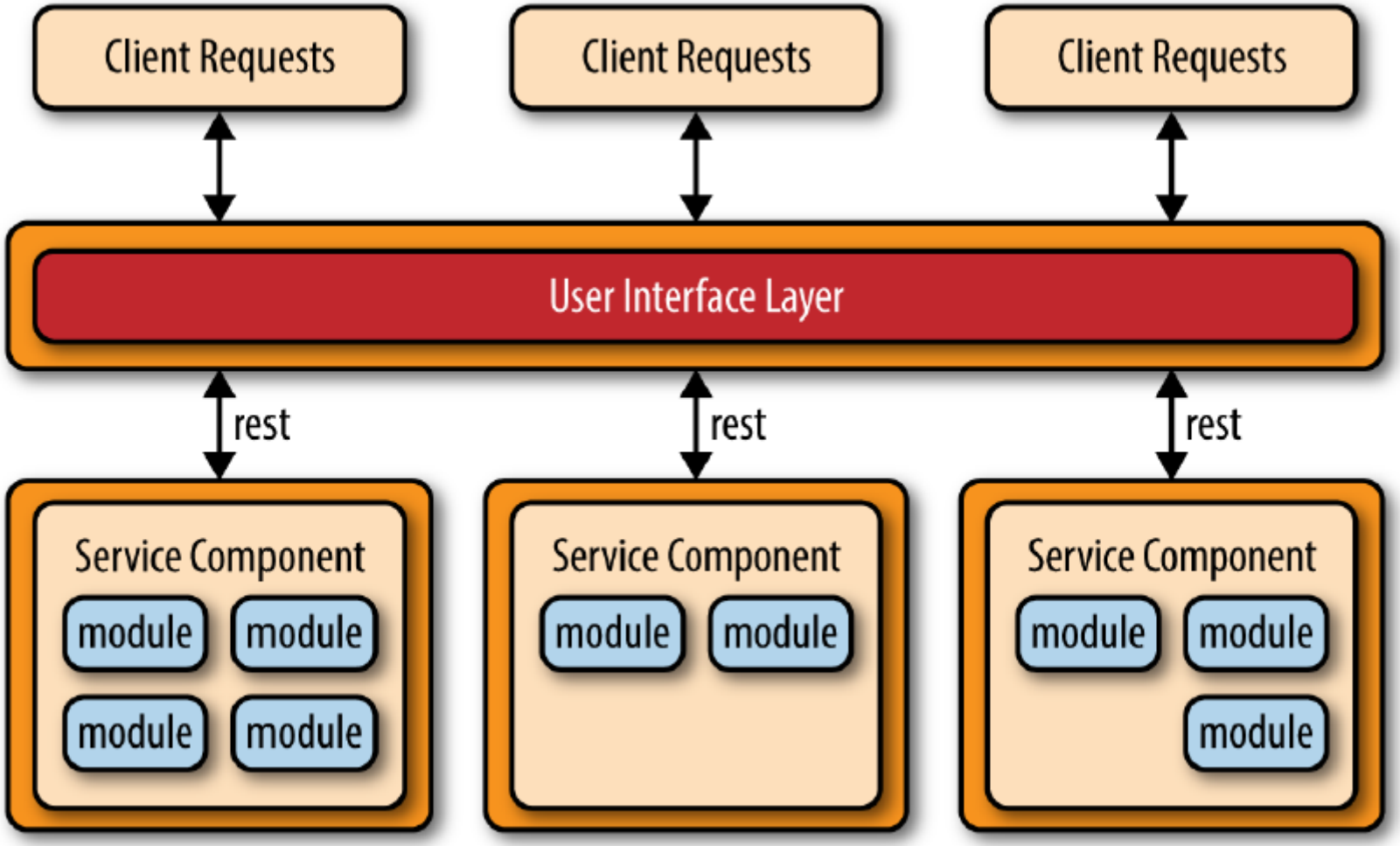
- El diseño de la granularidad es CLAVE. La idea final es EVITAR LA NECESIDAD DE ORQUESTACIÓN.
  - Si necesita “orquestación” es que el diseño de component services fue demasiado granular.

- Hay 3 grandes tipos:
  - API Rest: la aplicación es un conjunto de APIs orquestados
  - Application Rest: la aplicación tiene un cliente específico.
  - Mensajería centralizada: … muy parecida a un ESB.

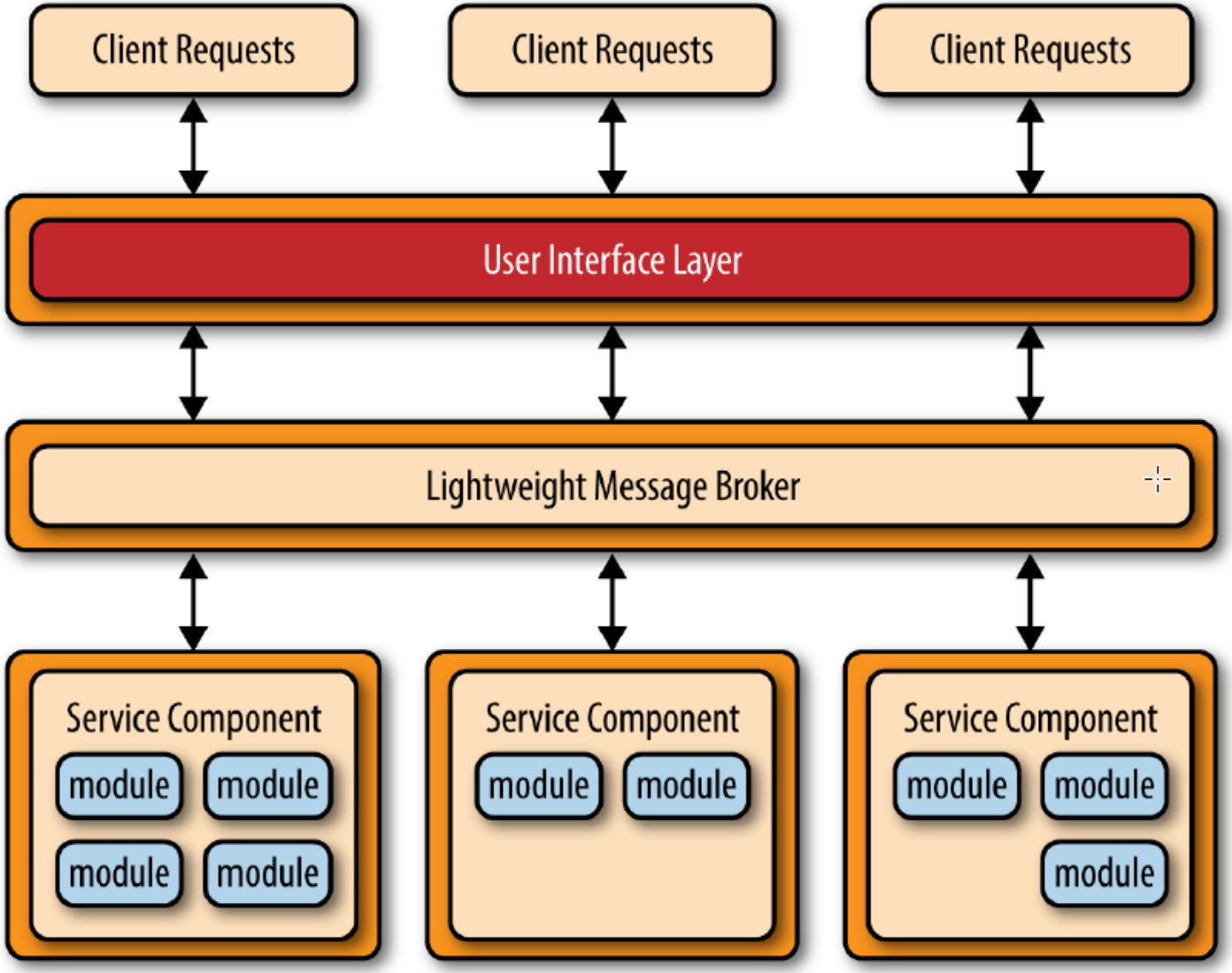
API Rest


Client Requests
Client Requests
Client Requests
Application Programming Interface (api)
rest
rest
rest
rest
rest
Service Component
module
module
Service Component
module
Service Component
module
Service Component
module
module
Service Component
module

Application Rest



Mensajería Centralizada



Arquitectura de Microservicios

- Es fácil tener equipos especializados por área (servicios) de negocio
- Es muy fácil hacer cambios y deploys aislados.
- Es fácil testear componentes.
- Puede tener problemas de rendimiento
- Proporciona mucha escalabilidad
- Es fácil especializar y tener programadores listos en una plataforma específica.

Arquitecturas basadas en Espacio

- Es un patrón orientado a resolver problemas de escalabilidad y concurrencia EXTREMOS.

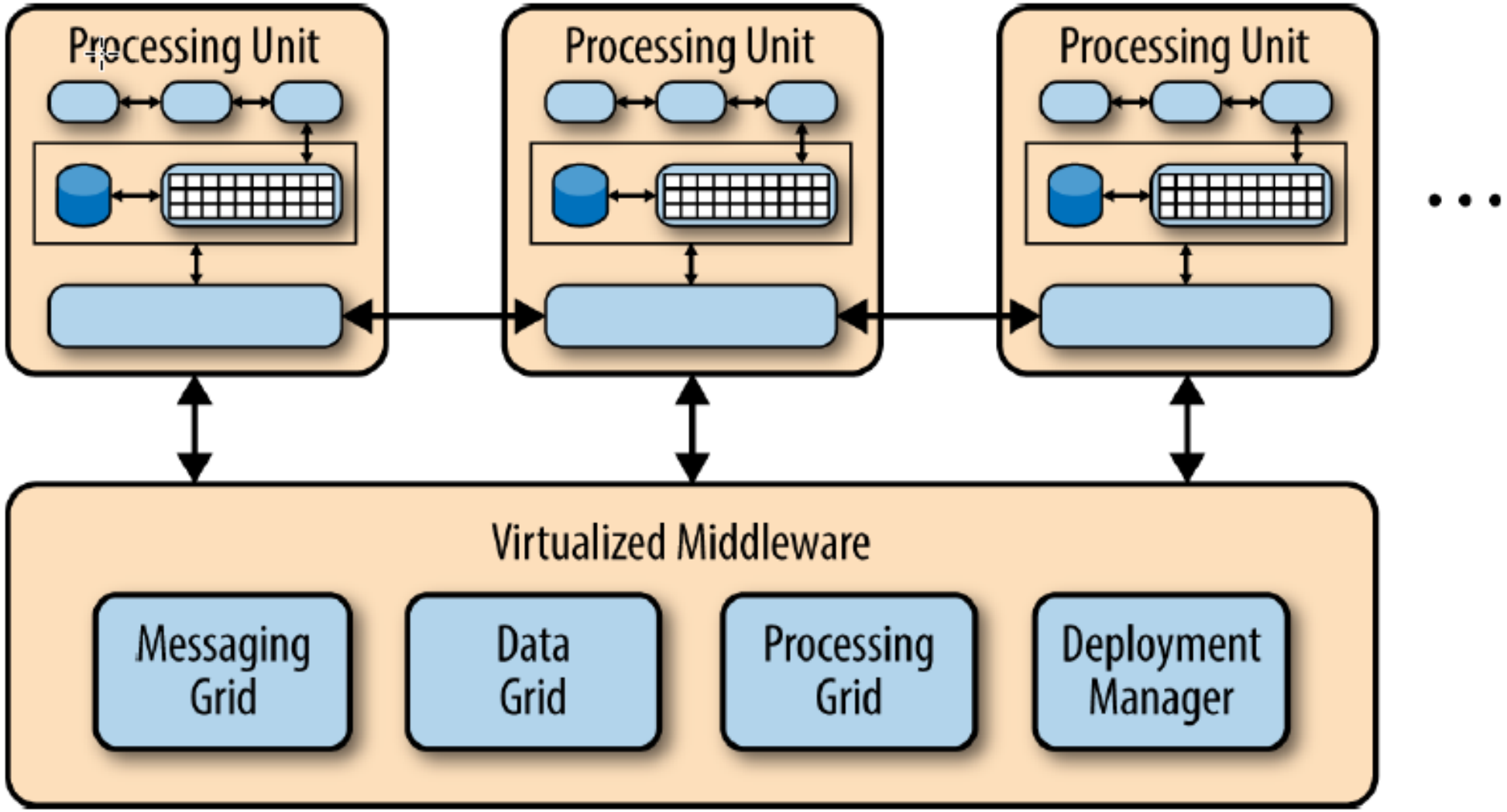
- La idea es MINIMIZAR los factores que limitan la escalabilidad. Se crean espacios virtuales de recursos.

- Se usa INTENSIVAMENTE LA MEMORIA RAM. Es posible levantar “ad hoc” más espacio, más procesadores, más espacio en disco cuando se requiere.

Aplicaciones basadas en espacio

- Hay 2 componentes primarios:
  - Unidad de procesamiento: es un componente que contiene la aplicación o parte de ella. Se pueden agregar tantas de estas como sea posible
  - Middleware virtualizado: se encarga de las comunicaciones y de sincronización de datos entre las unidades de procesamiento.

- La memoria de cada unidad de procesamiento puede “crecer” de forma “natural” usando el middleware virtual.


Processing Unit
Processing Unit
Processing Unit
Virtualized Middleware
Messaging Grid
Data Grid
Processing Grid
Deployment Manager

Aplicaciones basadas en espacio

- Son complejas y caras.
- Es fácil adaptar cambios de componentes, pero desarrollarlos y probar cosas es difícil.
- Ofrece alta escalabilidad y rendimientos.