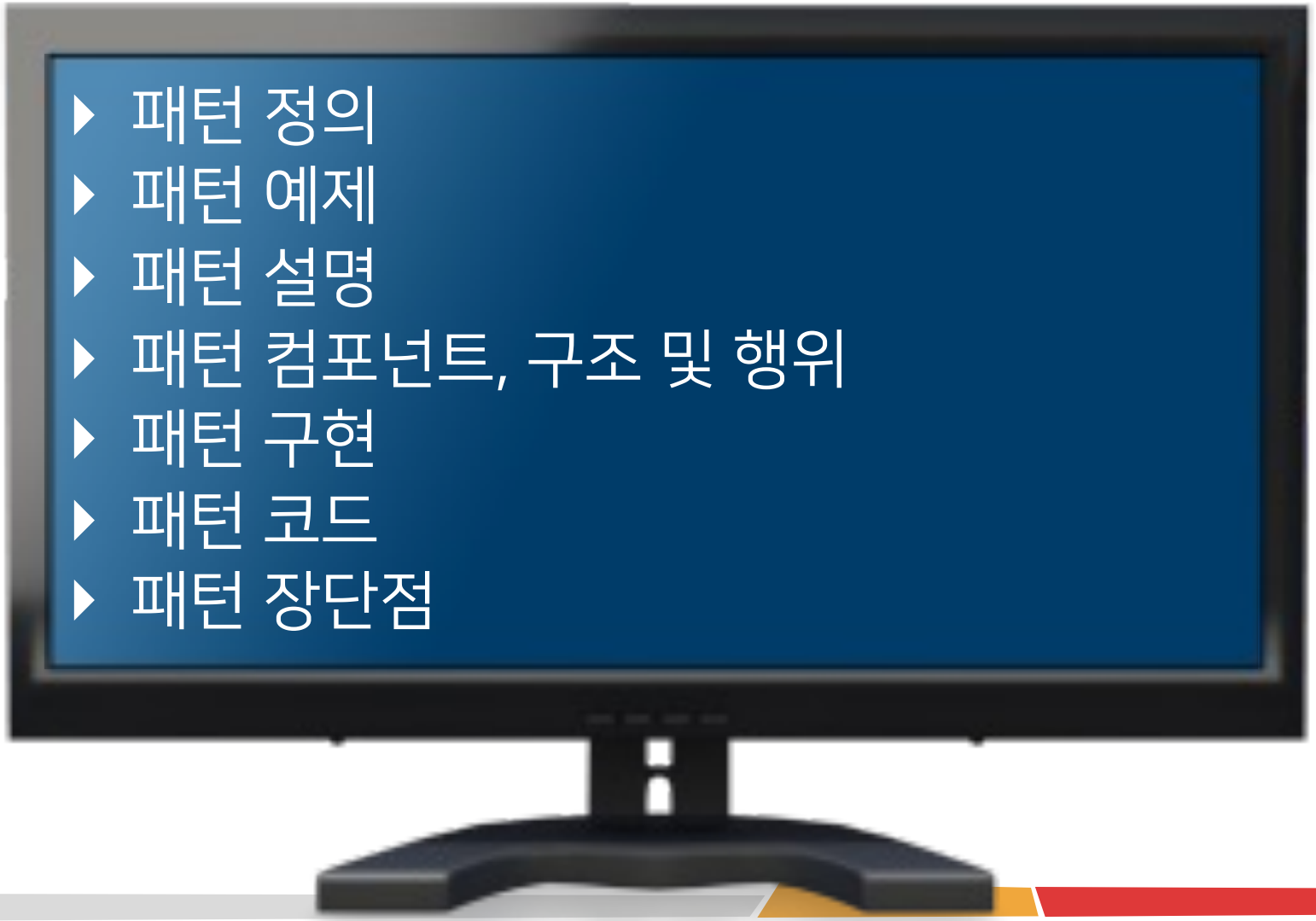
Spring 2023

소프트웨어 아키텍처 패턴: Client-Dispatcher-Server

Seonah Lee

Gyeongsang National University

Client-Dispatcher-Server 패턴

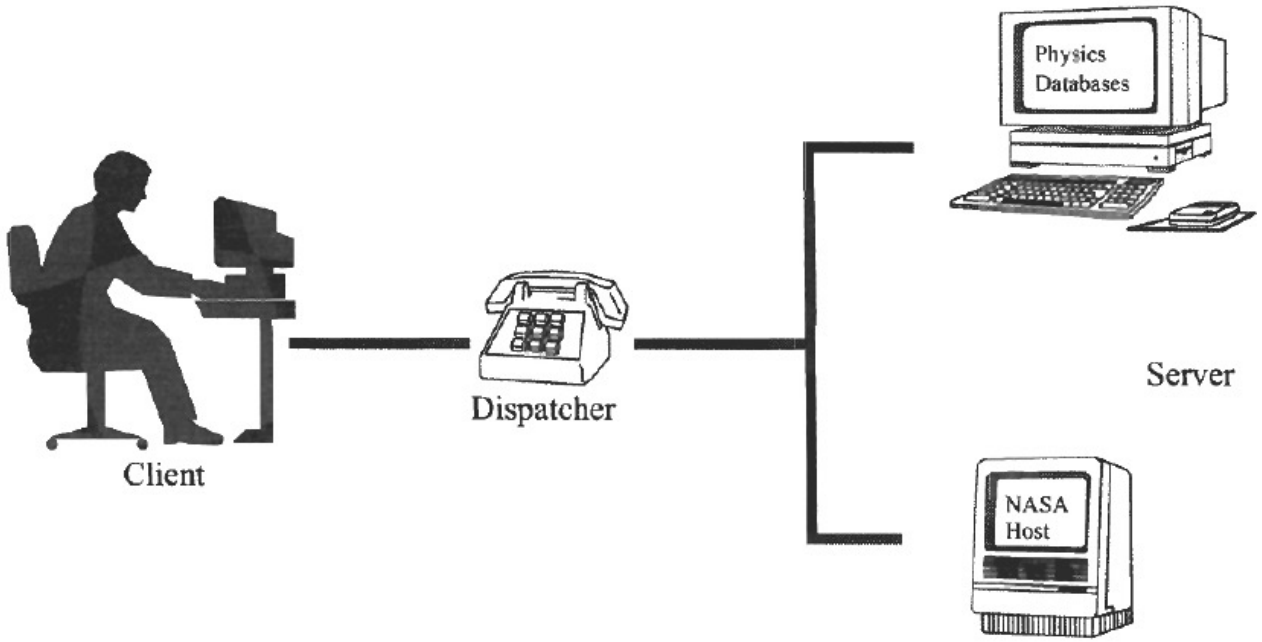


Dispatcher Pattern: Definition

- 정의
  - 클라이언트와 서버 간에 디스패처(**Dispatcher**) 중간 레이어 도입
    - 네임 서비스(**name service**)를 통해 위치 투명성을 제공
    - 클라이언트와 서버 간의 통신을 위한 세부 구현을 숨김

Dispatcher Pattern: Example

- 새로운 과학 정보를 검색하는 아킬레스라는 소프트웨어 시스템
  - 개별 정보 제공자에 액세스하기 위해서 해당 위치와 실행할 서비스 지정
  - 정보 제공자는 요청을 받으면, 적절한 서비스를 실행하고 요청 받은 정보를 반환



Dispatcher Pattern: Description

- 정황 (**Context**)
  - 통합할 분산된 서버들은 로컬 혹은 네트워크 상에 분산되어 실행
  - 분산된 서버들을 통합해야 함

- 예제
  - 정보 제공자는 전 세계에 분산되어 있음
  - 정보 제공자에 액세스하기 위해서는 해당 위치와 실행되어야 할 서비스를 지정함

Dispatcher Pattern: Description

- 문제 (**Problem**)
  - 분산된 서버를 사용할 때, 서버들 간에 통신할 방법을 제공해야 함
  - 컴포넌트들 간의 연결은 통신을 시작하기 전에 설정
  - 컴포넌트들 간의 연결은 어떤 통신 기능을 제공하느냐에 좌우
  - 컴포넌트의 핵심 기능은 통신 메커니즘의 세부 구현으로부터 분리
  - 클라이언트는 서버의 위치를 알 필요가 없음
  - 서버의 위치가 변경되어도 상관 없음

Dispatcher Pattern: Description

- 해법 (Solution)
  - 서비스 제공자의 위치와 상관없이 서비스를 사용할 수 있어야 함
  - 서비스 소비자의 핵심 기능 구현 코드
    - 서비스 제공자로의 연결을 설정하기 위해 사용하는 코드와는 분리 되어야 함
  - 디스패처 (Dispatcher) 도입
    - 클라이언트와 서버 간의 중간 레이어 역할 수행
    - 네임 서비스(name service)를 구현함
      - 네임 서비스는 물리적 위치가 아니라 이름으로 서버를 참조하는 방식
      - 위치 투명성(location transparency)를 제공
    - 클라이언트와 서버 간의 통신 채널을 설정하는 책임을 맡음

Dispatcher Pattern: Components

Client
• 시스템 태스크를 구현 • 디스패처에게 서버와의 연결을 요청 • 서버의 서비스를 호출

Server
• 클라이언트에게 서비스를 제공 • 디스패처에 자신을 등록

Dispatcher
• 클라이언트와 서버 간의 통신 채널을 설정 • 서버를 찾음 • 서버를 등록 • 서버의 등록을 삭제

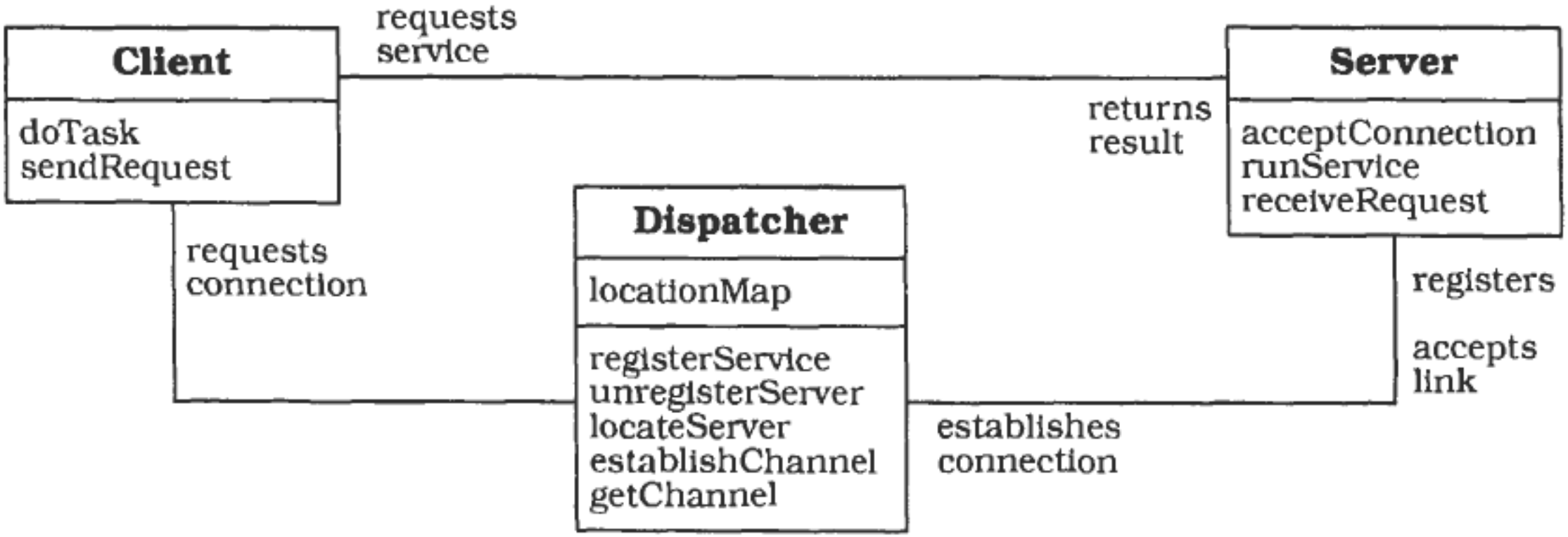
Dispatcher Pattern: Components

- **Client**
  - 특정 도메인을 위한 태스크를 수행
  - **Dispatcher** 에게 서버와의 연결을 요청
    - 서버와 연결 가능한 통신 채널이 있는지 확인
  - **Server**의 서비스를 호출
- **Server**
  - 클라이언트에게 여러 서비스를 제공
  - **Dispatcher**를 통해 자신을 등록시킴
    - 이름과 주소를 사용해 등록

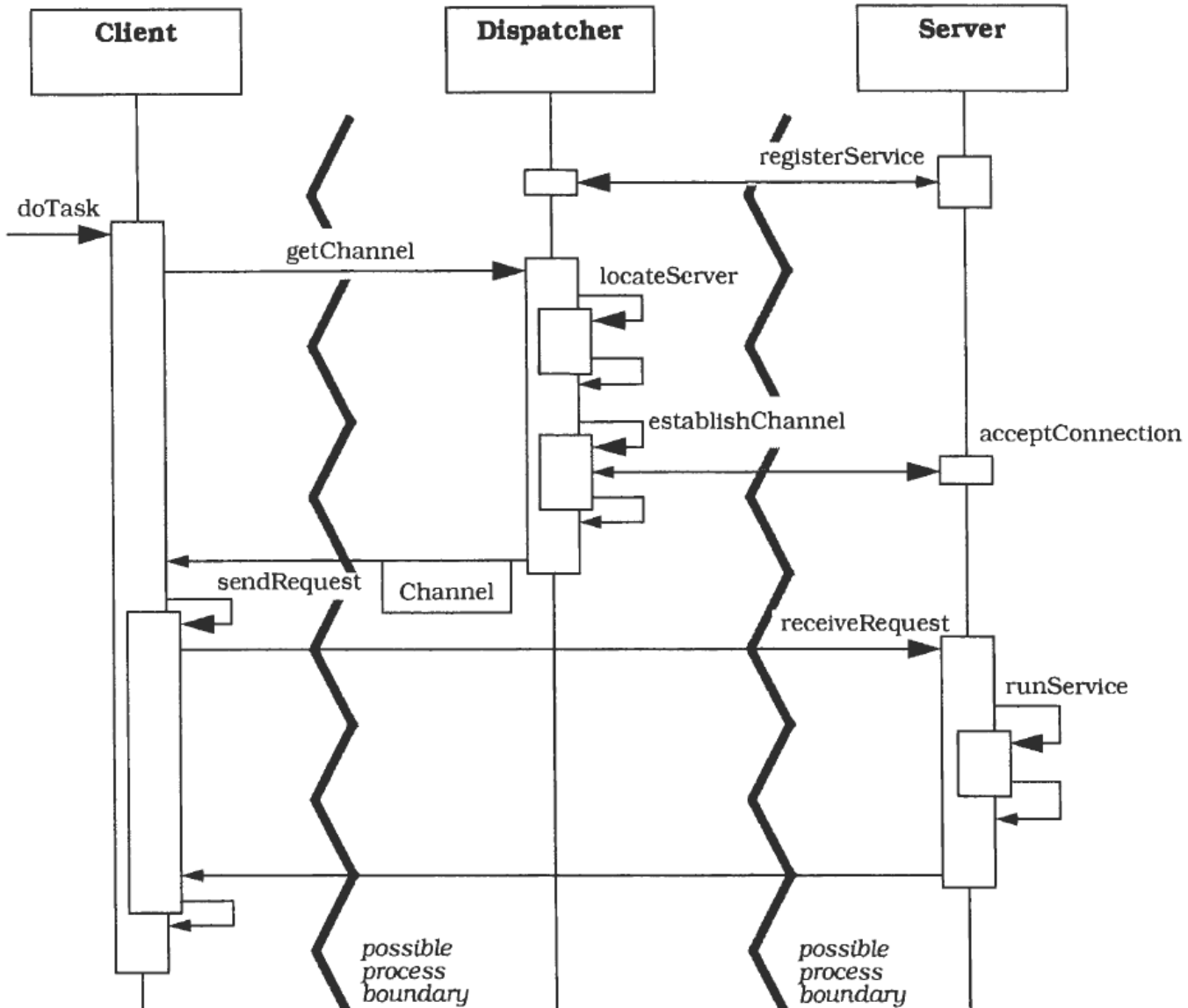
Dispatcher Pattern: Components

- **Dispatcher**
  - **Client**와 **Server**간의 통신 채널을 설정
    - 통신 메커니즘을 사용해 통신 링크를 설정
  - 서버를 찾음
  - 서버를 등록하거나 등록을 삭제
    - 서버의 이름을 서버의 물리적인 위치와 매핑

Dispatcher Pattern: Structure



Dispatcher Pattern: Behavior



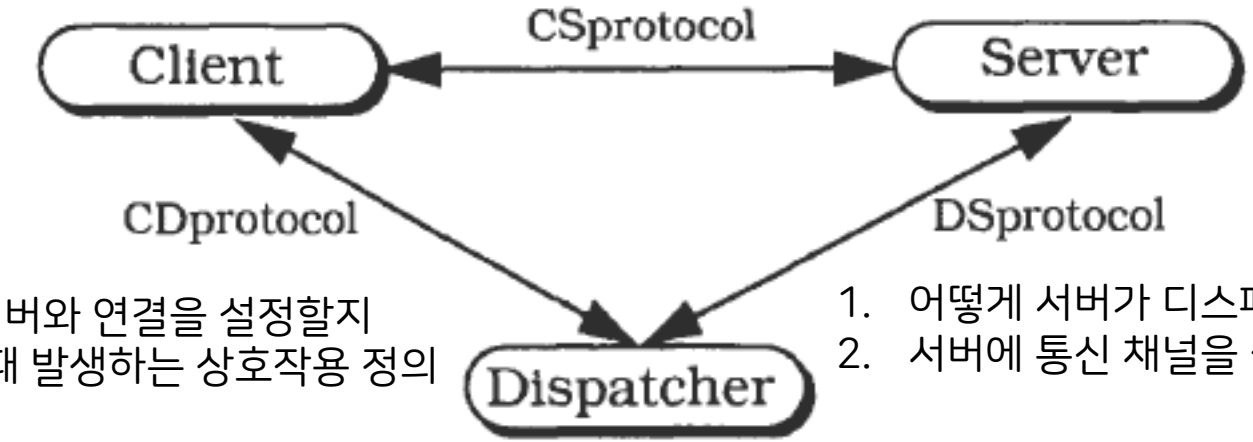
Dispatcher Pattern: Realization

- 구현 순서
  1. 애플리케이션을 서버 및 클라이언트로 분리한
     - 어떤 컴포넌트들을 서버로 구현하고 어떤 클라이언트들이 접근할 수 있는지?
  2. 어떤 통신 기능이 필요한지 파악
     - 클라이언트와 디스패처, 서버와 디스패처, 클라이언트와 서버 간의 상호 작용을 위한 통신 기능을 선택
       - 예: 동일한 머신 내에서는 공유 메모리를 사용하여 **IPC** 메커니즘의 빠른 방법 선택
       - 예: 다른 머신에 분산되어 있을 경우 소켓을 사용하여 통신

Dispatcher Pattern: Realization

- 구현 순서
  3. 컴포넌트 간의 상호작용 프로토콜을 정의
     - 각각은 서로 다른 종류의 프로토콜이 필요
       - 프로토콜은 통신 채널, 전송 메시지, 데이터 구조를 초기화하고 저장하는 일련의 활동 수행

1. 클라이언트와 서버가 어떻게 통신할지 정의



1. 클라이언트가 서버와 연결을 설정할지 디스패처에 요청할 때 발생하는 상호작용 정의

1. 어떻게 서버가 디스패처에 등록되는지 정의
2. 서버에 통신 채널을 설정할 때 필요한 동작 정의

Dispatcher Pattern: Realization

- 구현 순서
  4. 서버의 이름을 어떻게 지을 것인지 결정
     - 특정한 위치 정보를 구체적으로 드러내지 않는 이름 도입
       - 예: **IP** 주소는 물리적인 위치에 종속되므로 바람직하지 않음
  5. 디스패처를 설계하고 구현
     - 디스패처가 클라이언트의 주소 공간 내 위치 시: 로컬 프로시저 사용
     - 디스패처가 클라이언트의 주소 공간 내가 아니면: **TCP**, 혹은 공유 메모리 등 사용
     - 병목 현상을 고려한 성능 문제도 고려할 수 있음
  6. 디스패처 인터페이스에 맞게 클라이언트 컴포넌트 및 서버 컴포넌트를 구현

Dispatcher Pattern: Implementation

```
public class CDS {
     public static Dispatcher disp = new Dispatcher();
     public static void main(String args[]) {
          Service s1 = new PrintService("printSvc","srv1");
          Service s2 = new PrintService("printSvc","srv2");
          Client client = new Client();
          client.doTask();
     }
}
```

Dispatcher Pattern: Implementation

```
import java.util.*;
import java.io.*;

// Exception thrown by the dispatcher:
class NotFound extends Exception {}
class Dispatcher {
     Hashtable registry = new Hashtable();
     Random rnd = new Random(123456); // for random access

     public void register (String svc, Service obj) {
          Vector v = (Vector) registry.get(svc);
          if (v == null) {
               v = new Vector();
               registry.put(svc, v);
          }
          v.addElement(obj);
     }
     public Service locate(String svc) throws NotFound {
          Vector v = (Vector) registry.get(svc);
          if (v == null) throw new NotFound();
          if (v.size() == 0) throw new NotFound();
          int i = rnd.nextInt() % v.size();
          return (Service) v.elementAt(i);
     }
}
```

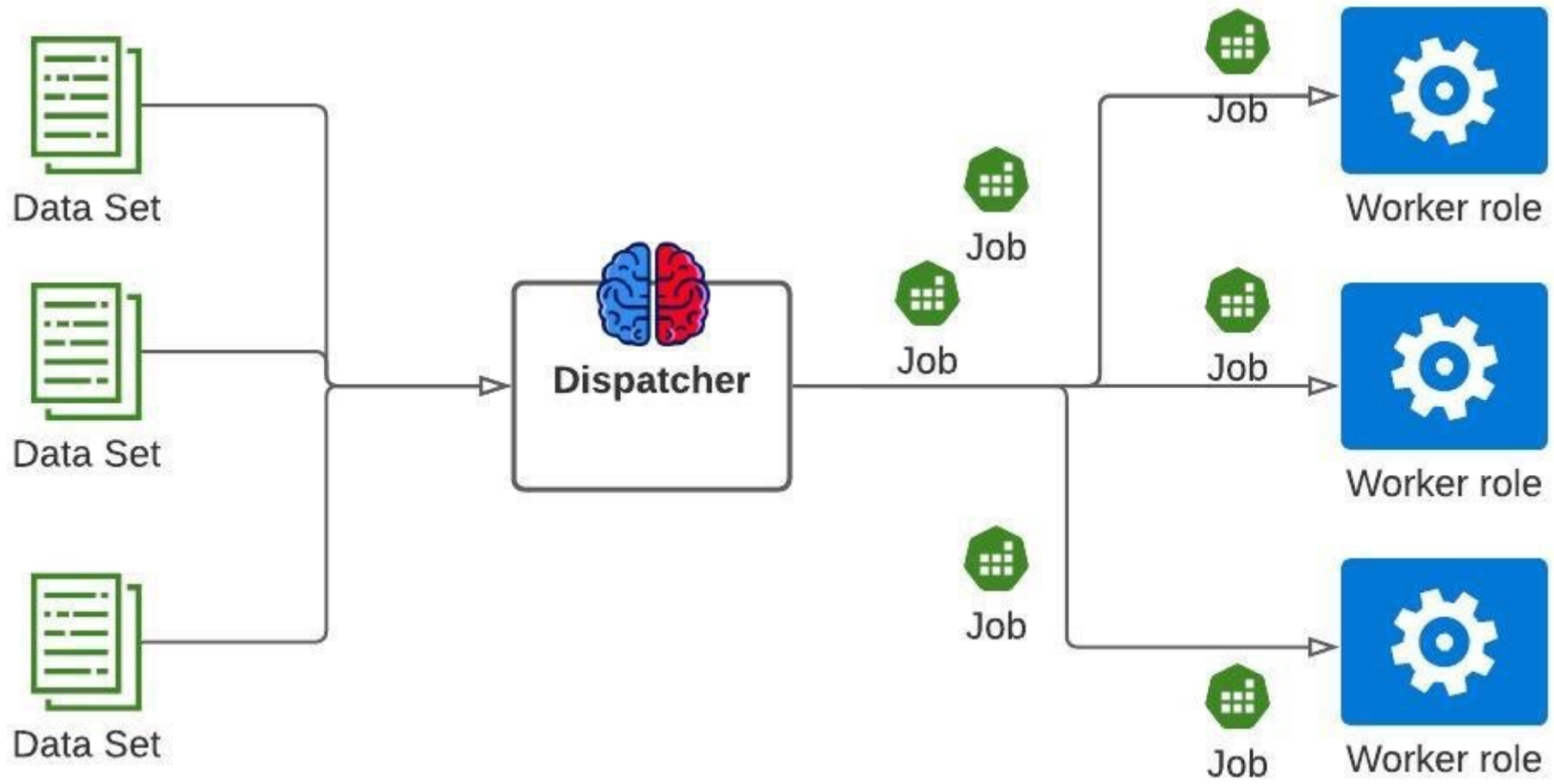
Dispatcher Pattern: Implementation

```
abstract class Service {
     String nameOfService; // service name
     String nameOfServer; // server name
     public Service(String svc, String srv) {
          nameOfService = svc;
          nameOfServer  = srv;
          CDS.disp.register(nameOfService,this);
     }
     abstract public void service(); // service provided
}
class PrintService extends Service {
     public PrintService(String svc, String srv) {
          super(svc,srv);
     }
     public void service() { // test output
          System.out.println("Service " + nameOfService
                    + " by " + nameOfServer);
          // here the service code would be implemented
     }
}
```

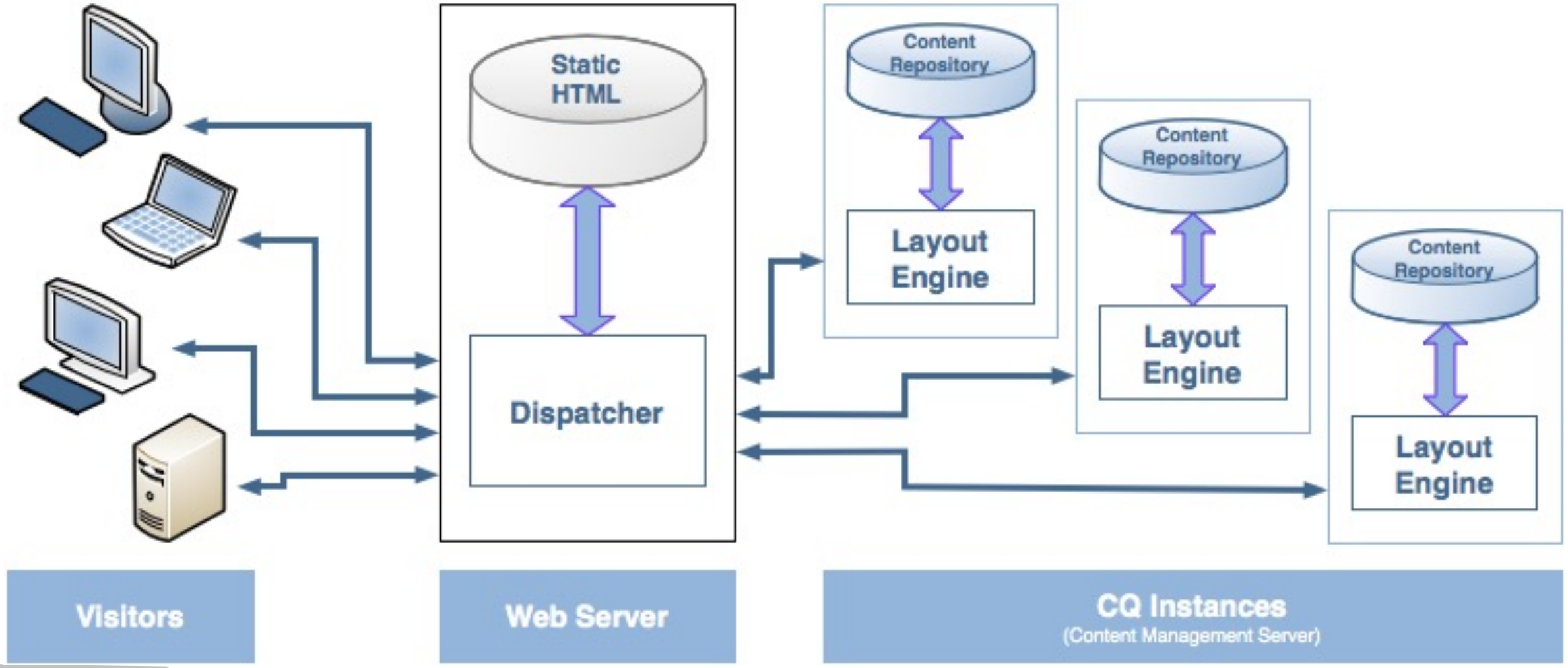
Dispatcher Pattern: Implementation

```
class Client {
     public void doTask()
     {    Service s;
          try { s = CDS.disp.locate("printSvc");
                s.service();
          }
          catch (NotFound n) {
                System.out.println("Not available");
          }
          try { s = CDS.disp.locate("printSvc");
                s.service();
          }
          catch (NotFound n) {
                System.out.println("Not available");
          }
          try { s = CDS.disp.locate("drawSvc");
                s.service();
          }
          catch (NotFound n) {
                System.out.println("Not available");
          }
     }
}
```

Dispatcher Pattern: Case Studies

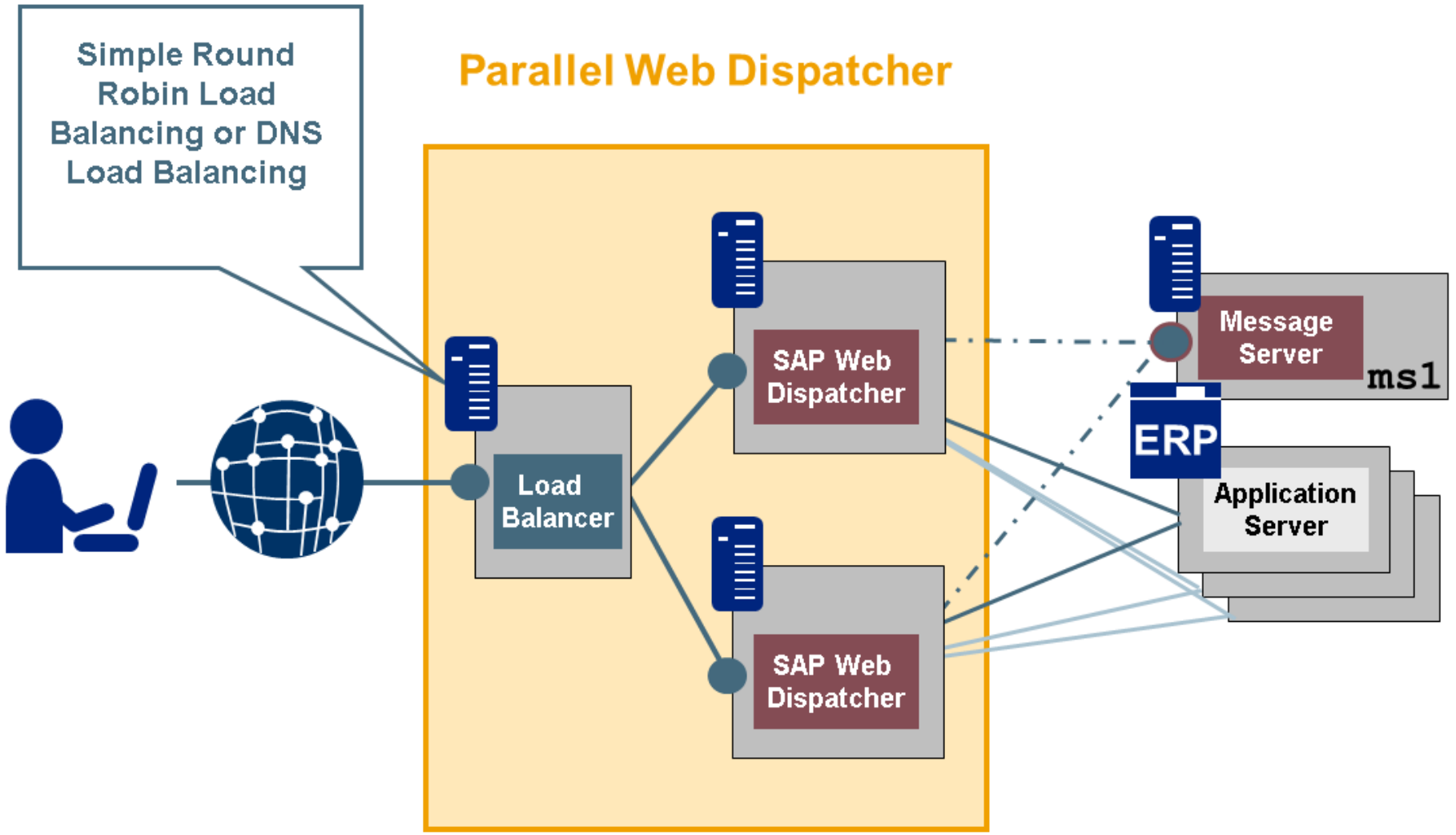


Dispatcher Pattern: Case Studies



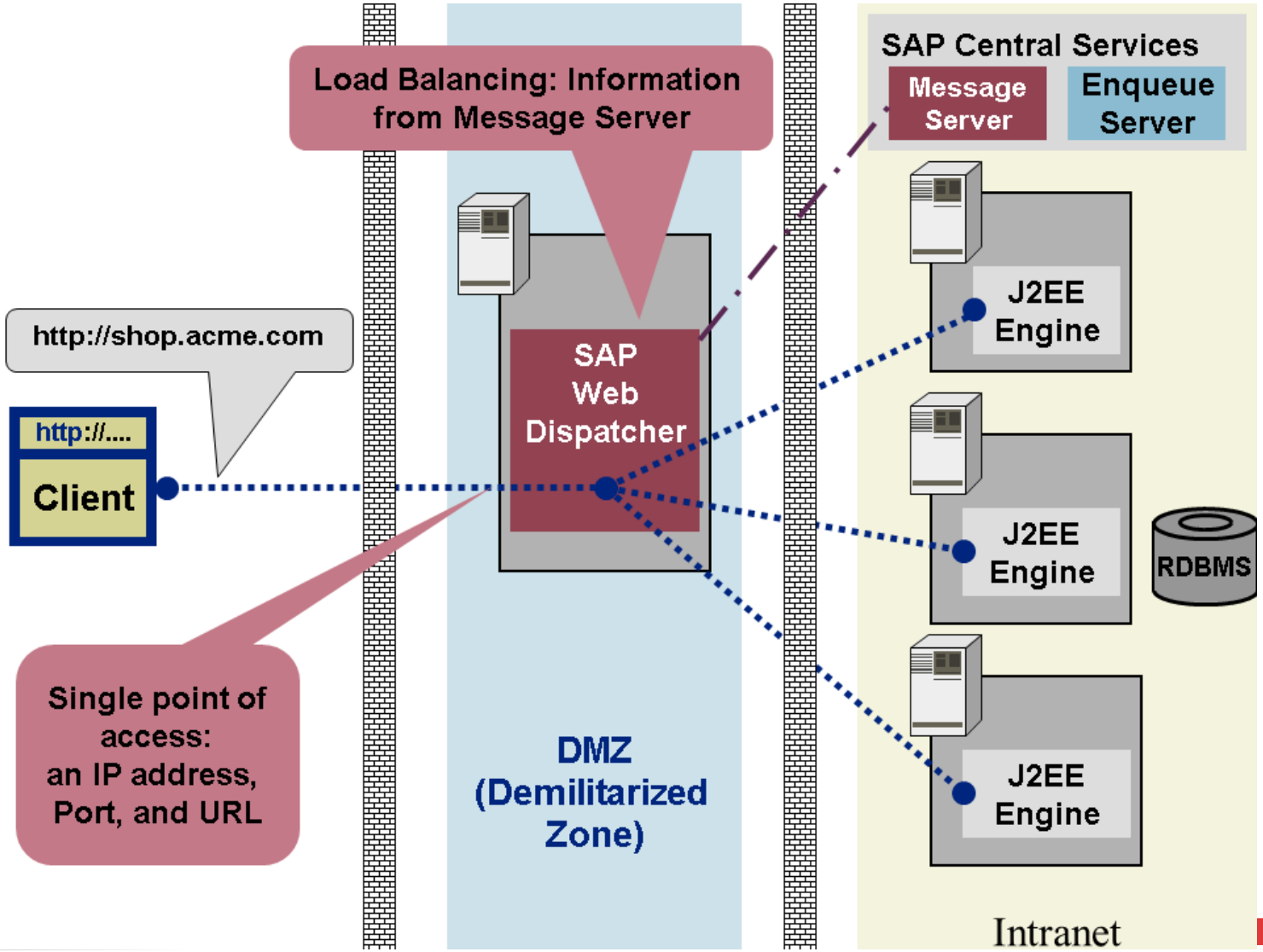
https://experienceleague.adobe.com/docs/experience-manager-dispatcher/using/dispatcher.html?lang=en

Dispatcher Pattern: Case Studies

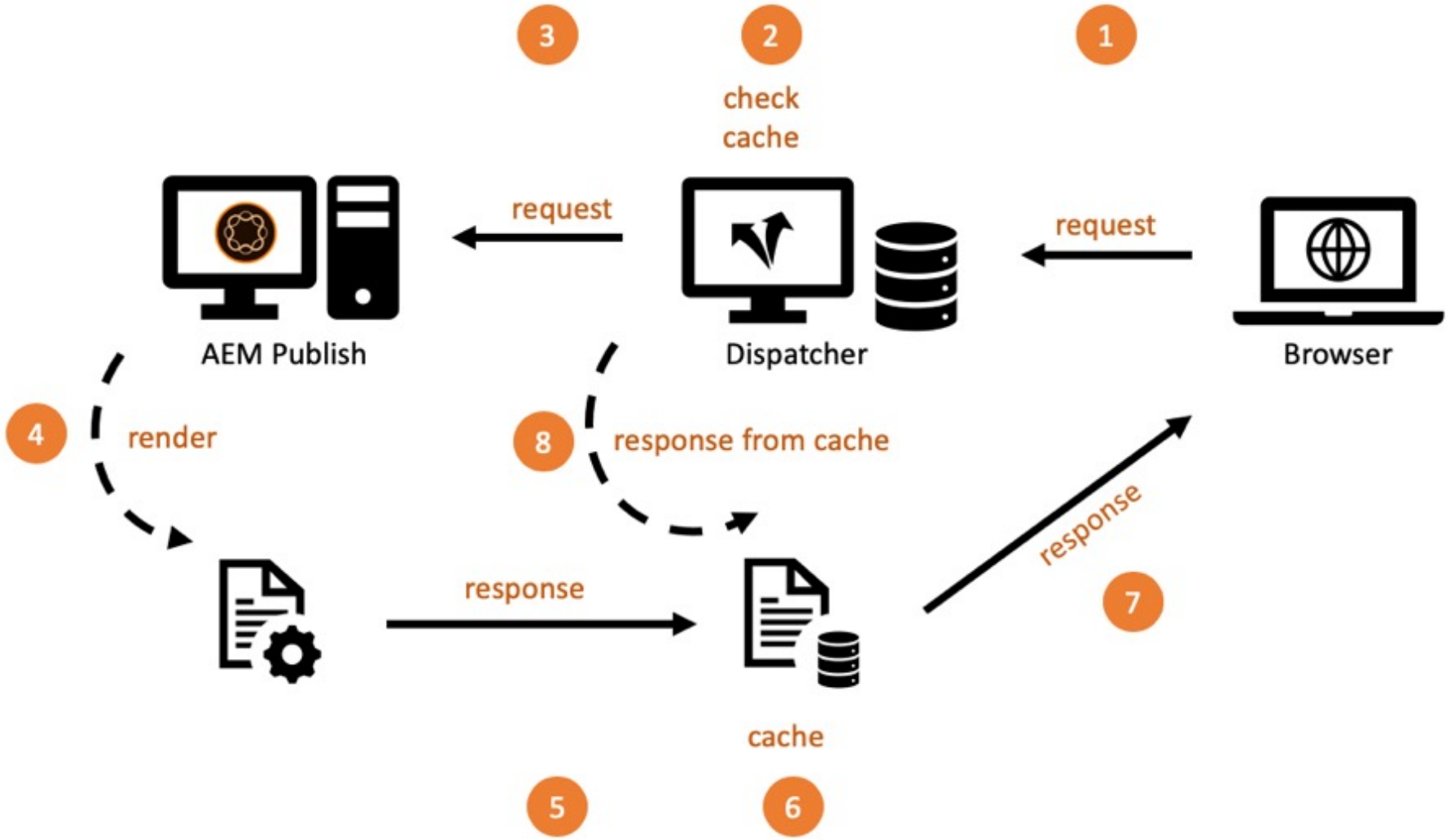


https://help.sap.com/doc/saphelp_nw74/7.4.16/en-us/48/9a9a6b48c673e8e10000000a42189b/frameset.htm

Dispatcher Pattern: Case Studies



Dispatcher Pattern: Case Studies



1.A user requests a page
2.The Dispatcher checks, if it already has a rendered version of that page.
3.The Dispatcher requests the page from the Publish system
4.On the Publish system, the page is rendered by a JSP or an HTL template
5.The page is returned to the Dispatcher
6.The Dispatcher caches the page
7.The Dispatcher returns the page to the browser
8.If the same page is requested a second time, it can be served directly from the Dispatcher cache.

Dispatcher Pattern: Benefits

- 서버의 교환가능성 보장
  - 서버를 변경하거나 추가할 때 **Dispatcher**와 **Client**를 변경할 필요가 없음
- 서버의 위치 투명성 보장
  - **Client**는 서버가 어디에 위하는지 알 필요 없음, 서버가 다른 하드웨어로 이동 가능
- 서버의 재구성 가능
  - 개발자는 네트워크의 어떤 서버를 실행할지 결정을 지연할 수 있음
  - 해당 패턴은 소프트웨어 시스템을 분산 시스템으로 만드는 것을 준비하도록 도움
- 장애 허용성 보장
  - 네트워크 혹은 서버의 실패가 발생 시, 다른 네트워크의 서버를 활성화할 수 있음

Dispatcher Pattern: Liabilities

- 명시적이며 우회적인 연결을 설정하기 때문에 효율성이 낮음
  - 해당 패턴의 성능은 **Dispatcher**에서 도입되는 오버헤드에 의존
- 디스패처 컴포넌트의 인터페이스에 발생하는 변경에 민감함
  - **Dispatcher**가 중심 역할을 함
  - 소프트웨어 시스템인 **Dispatcher**의 **Interface**의 변경에 민감함

Question?

Seonah Lee
saleese@gmail.com