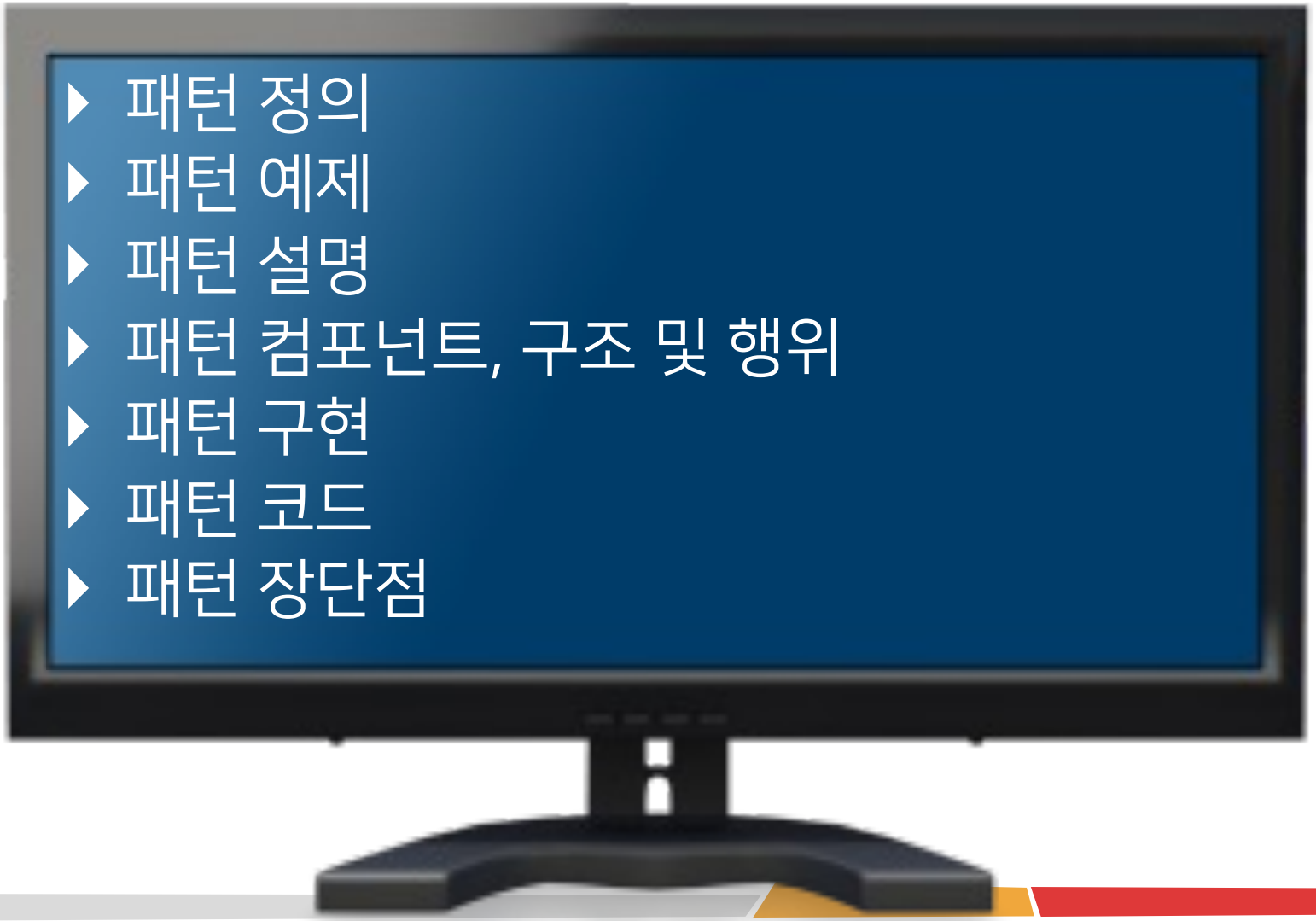
Spring 2023

소프트웨어 아키텍처 패턴: Broker

Seonah Lee

Gyeongsang National University

Broker 패턴

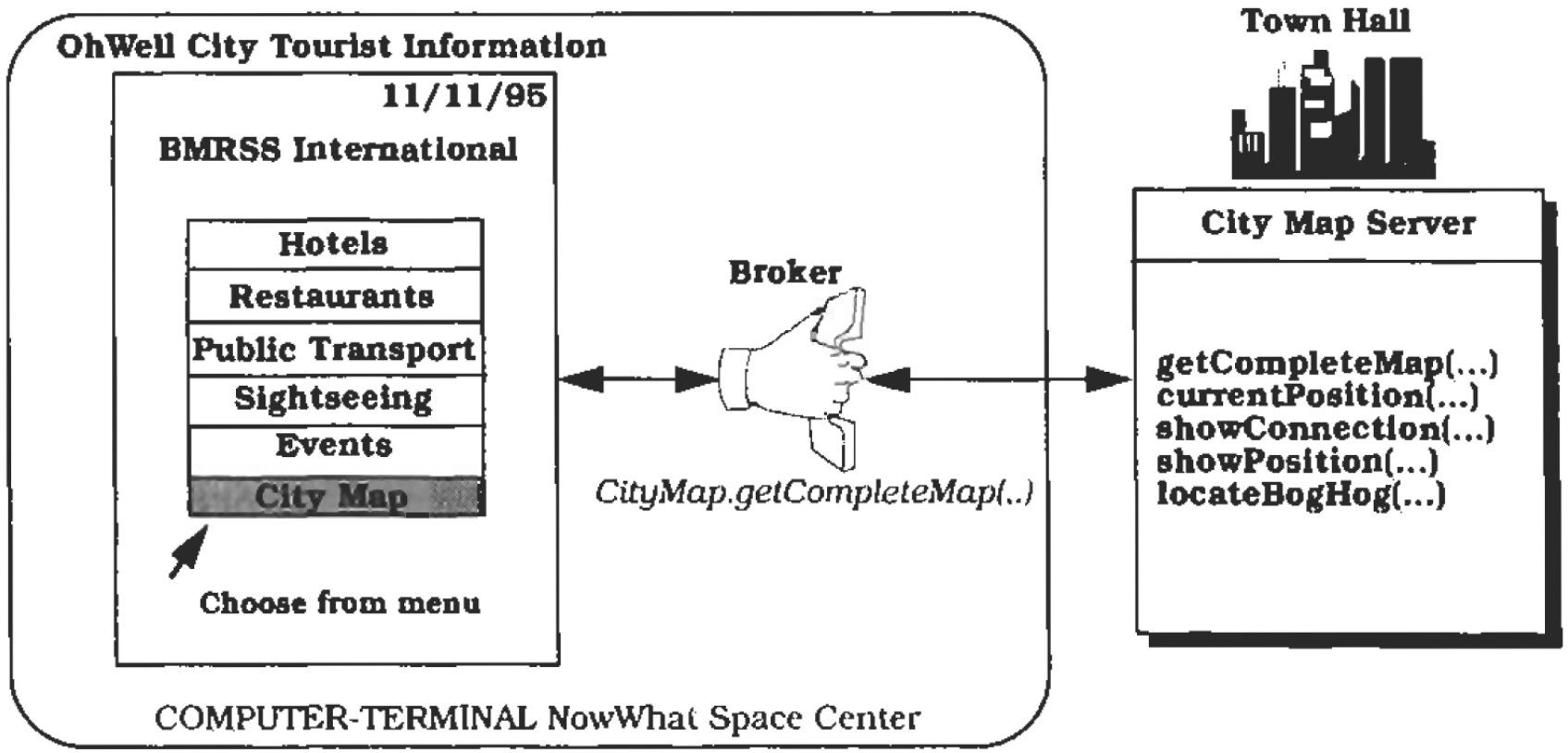


Broker Pattern: Definition

- 정의
  - 외부에 분산된 컴포넌트를 호출하려고 할 때 클라이언트 요청 값을 분석하여 서버 컴포넌트에 전달하고 그 결과값을 전달하는 역할을 하는 패턴
  - 클라이언트와 서버 사이의 브로커라는 컴포넌트를 두어 보다 효과적으로 서버와 클라이언트 사이를 분리할 수 있어 분산 시스템을 구축하는데 용이함

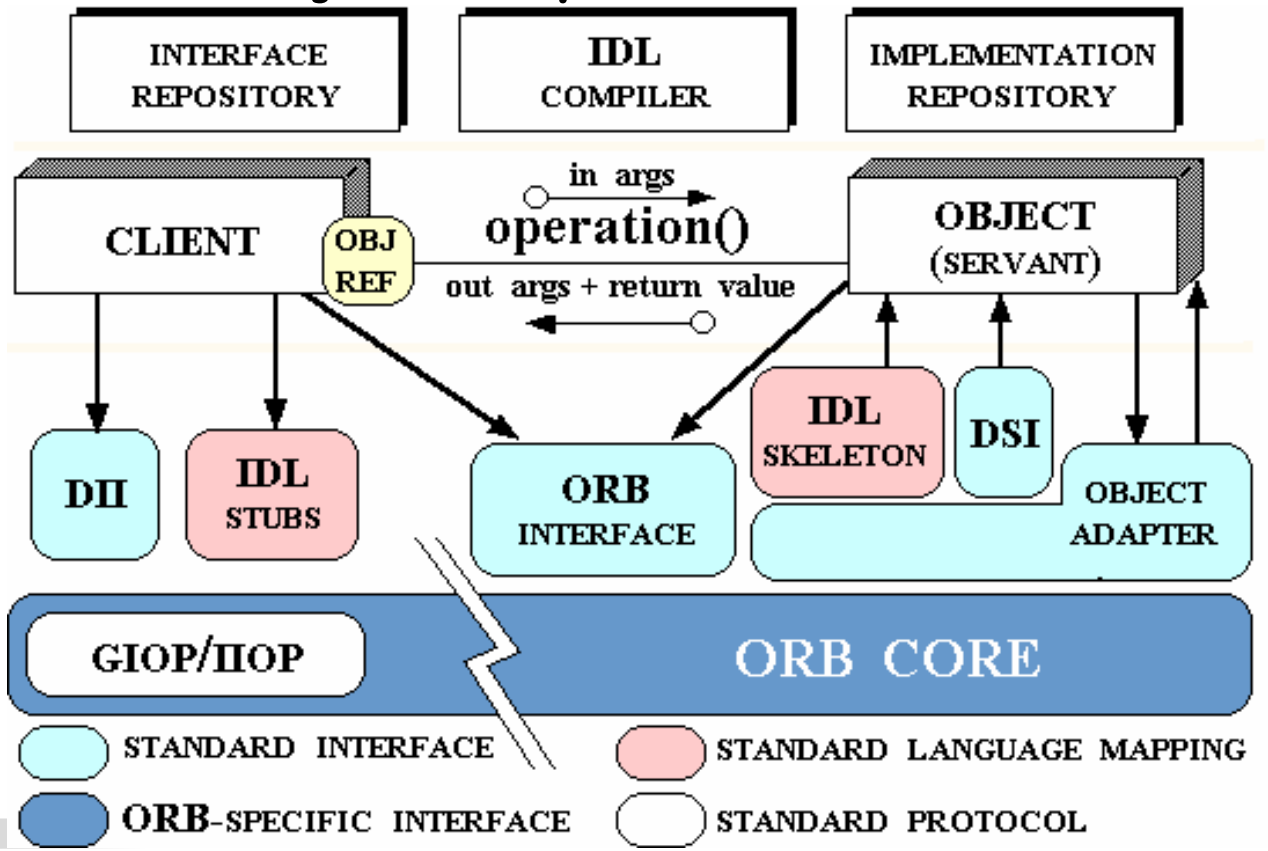
Broker Pattern: Example

- 예제
  - 광역 네트워크 기반의 **CIS(city information system)** 시스템

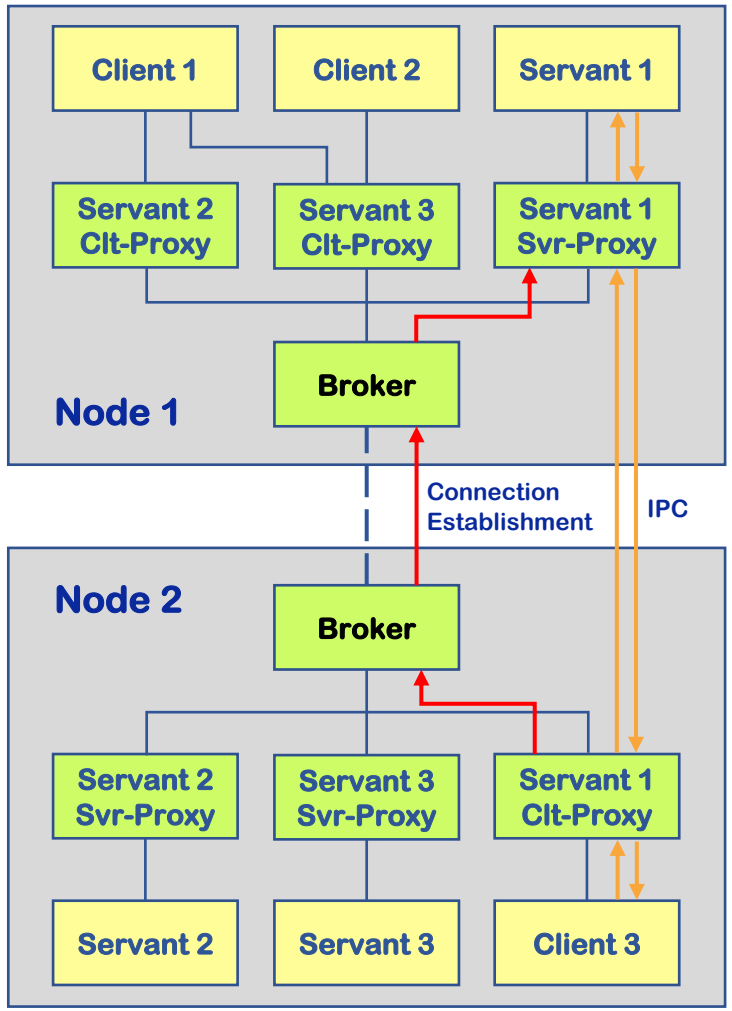


Broker Pattern: Example

- 예제
  - CORBA (Common Object Request Broker Architecture)



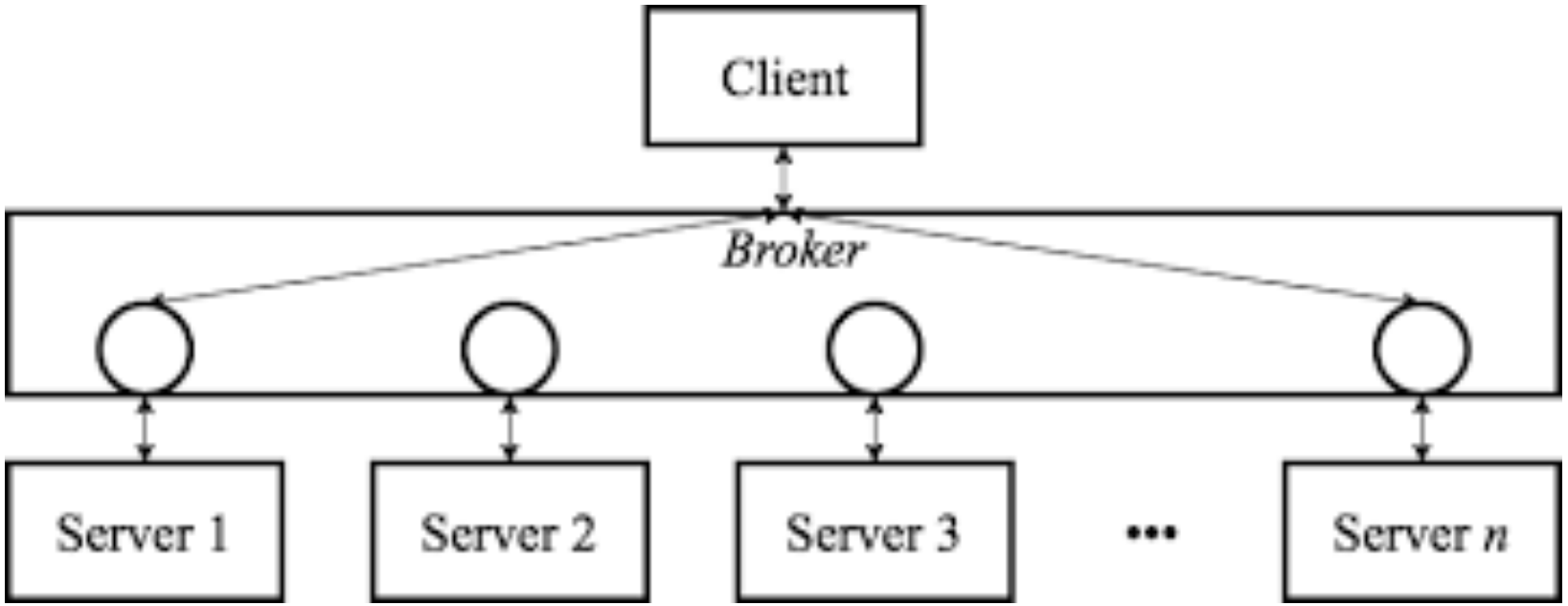
Broker Pattern



- 분산 컴포넌트의 원격 제공 및 위치투명 **(location-transparent)**한 서비스 호출을 통해 다른 컴포넌트 서비스에 엑세스
- 런타임에 컴포넌트 교환, 추가, 제거 가능
- 아키텍처는 특정 시스템 혹은 특정 구현의 세부 사항을 컴포넌트나 서비스 사용자로부터 숨겨야 함

Broker Pattern: Description

- 정황(Context)
  - 독립적인 컴포넌트 형태로 이질적인 환경에서 작동하는 분산 시스템을 개발하는 경우



https://www.researchgate.net/figure/The-Broker-architectural-pattern_fig2_330079838

Broker Pattern: Description

- 문제(**Problem**)
  - 독립 컴포넌트마다 실행 환경이 다르고 이들끼리 통신이 필요한 경우
    - **IPC(Inter-process communication** 프로세스간 통신)가 가능하도록 지원
    - 제약 사항 발생
      - 통신 메커니즘 종속
      - 클라이언트가 서버의 위치를 반드시 파악
      - 단일 프로그래밍 언어 등
  - 클라이언트와 서버들이 추가, 삭제 및 변경이 자주 일어날 경우
    - 컴포넌트를 추가, 제거, 교환, 활성화, 찾기를 위한 서비스 필요
      - 이식성과 상호운용성을 보장하기 위해 특정 시스템의 세부 구현에 의존하지 않아야 함

Broker Pattern: Description

- 해법(Solution)
  - **Broker** 컴포넌트를 도입하여 클라이언트와 서버 사이를 분리
  - **Broker** 컴포넌트가 클라이언트와 서버의 정보를 가지고 있어 서로간의 통신을 조율
    - 서버는 자신을 브로커에 등록, 자신의 서비스를 사용할 수 있도록 메서드의 인터페이스를 제공
    - 클라이언트는 브로커를 통해 요청**(request)**를 보내 서버의 서비스에 접근
    - 브로커는 서버 찾기, 서버 요청, 클라이언트에 결과와 예외 전송하는 작업 수행
  - 클라이언트와 서버 **Proxy**를 두어 특정 환경과 관련된 부분을 처리
  - **Bridge**를 두어 네트워크 통신과 관련된 부분을 이관하여 처리
  - 분산 기술 **+** 객체 기술: 적절한 객체에 메시지 호출을 보내 분산 서비스에 접근 가능

Broker Pattern: Components

Client

- 사용자 기능 구현
- 클라이언트측 프록시를 통해 서버에 요청 전송

Broker

- 서버 등록, 등록에서 제거
- 서버의 위치를 찾음
- **API** 제공
- 메시지 전송
- 오류 복구
- **Bridge** 컴포넌트를 통해 다른 브로커들과 상호작용

Server

- 서비스 구현
- 로컬 브로커에 자신을 등록
- 서버측 프록시를 통해 클라이언트에 돌려보낼 응답 및 예외 전송

Client-side Proxy

- 특정 시스템에 맞춰진 기능 캡슐화
- 클라이언트와 브로커 사이를 중재

Bridge

- 특정 네트워크에 맞춰진 기능 캡슐화
- 원격 브로커의 브리지와 로컬 브로커 사이를 중재

Server-side Proxy

- 서버 내 서비스 호출
- 특정 시스템에 맞춰진 기능 캡슐화
- 서버와 브로커 사이를 중재

Broker Pattern: Components

- **Client**
  - 서버에서 제공하는 서비스들에 접근하는 애플리케이션
  - 원격 서비스를 호출하기 위해 브로커에 요청을 보냄
- **Server**
  - 서비스 구현
    - 오퍼레이션과 속성으로 구형된 인터페이스들을 통해 기능을 제공하는 객체 구현
      - **(IDL** 혹은 바이너리 표준 방식**)**
  - 로컬 브로커에 자신을 등록
  - 브로커를 통해 클라이언트에 돌려보낼 응답과 예외 전송

Broker Pattern: Components

- **Broker**
  - 클라이언트에서 서버로 요청 + 응답과 예외를 클라이언트로 전송
    - 시스템 식별자에 근거하여 요청을 받을 수신자의 위치를 찾는 방법을 구현
    - 서버를 등록, 등록에서 제거, 서버 위치를 검색하는 오퍼레이션 제공
    - 서버의 메소드 호출을 위한 오퍼레이션 관련 **API**를 클라이언트와 서버에 제공
    - 메시지를 전송하고 오류를 복구함
  - **Bridge** 컴포넌트를 통해 다른 브로커들과 상호작용

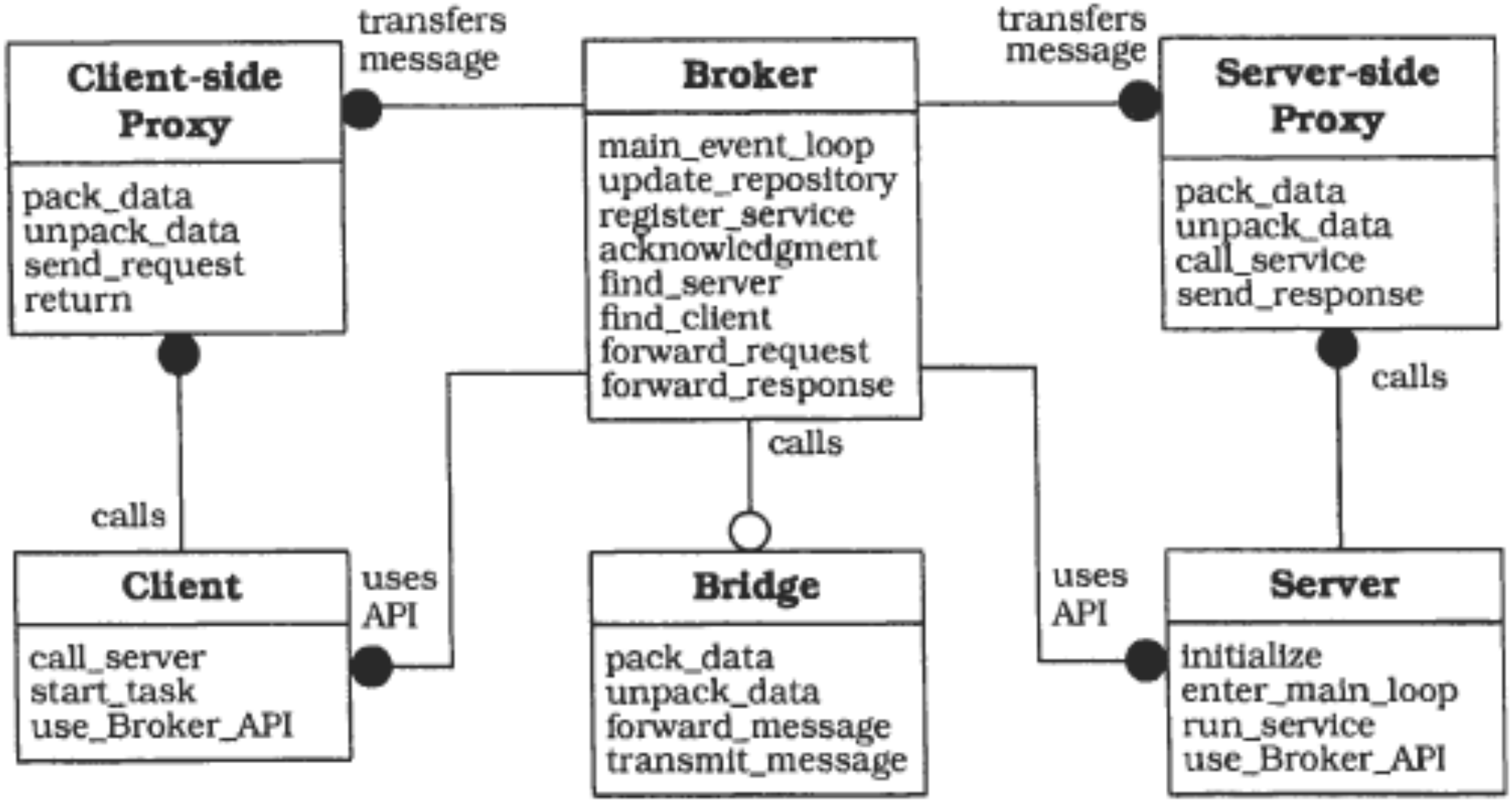
Broker Pattern: Components

- **Client-side Proxy**
  - 특정 시스템에 맞춰진 기능을 캡슐화
  - 클라이언트와 브로커 사이를 중재, 그 사이에서 레이어 역할을 수행
    - 해당 프록시로 인하여 클라이언트는 원격 객체를 로컬 객체로 간주
  - 클라이언트로부터 다음 세부 구현을 숨김
    - 클라이언트와 브로커 간의 메시지 전송을 위해 사용되는 **IPC(Inter-process communication)** 메커니즘
    - 메모리 블록의 생성과 제거
    - 매개변수의 결과의 마샬링
  - **Broker** 패턴의 일부분으로 특수하게 정의된 객체 모델을 클라이언트를 구현하기 위해 사용된 프로그램 언어의 객체 모델로 변환

Broker Pattern: Components

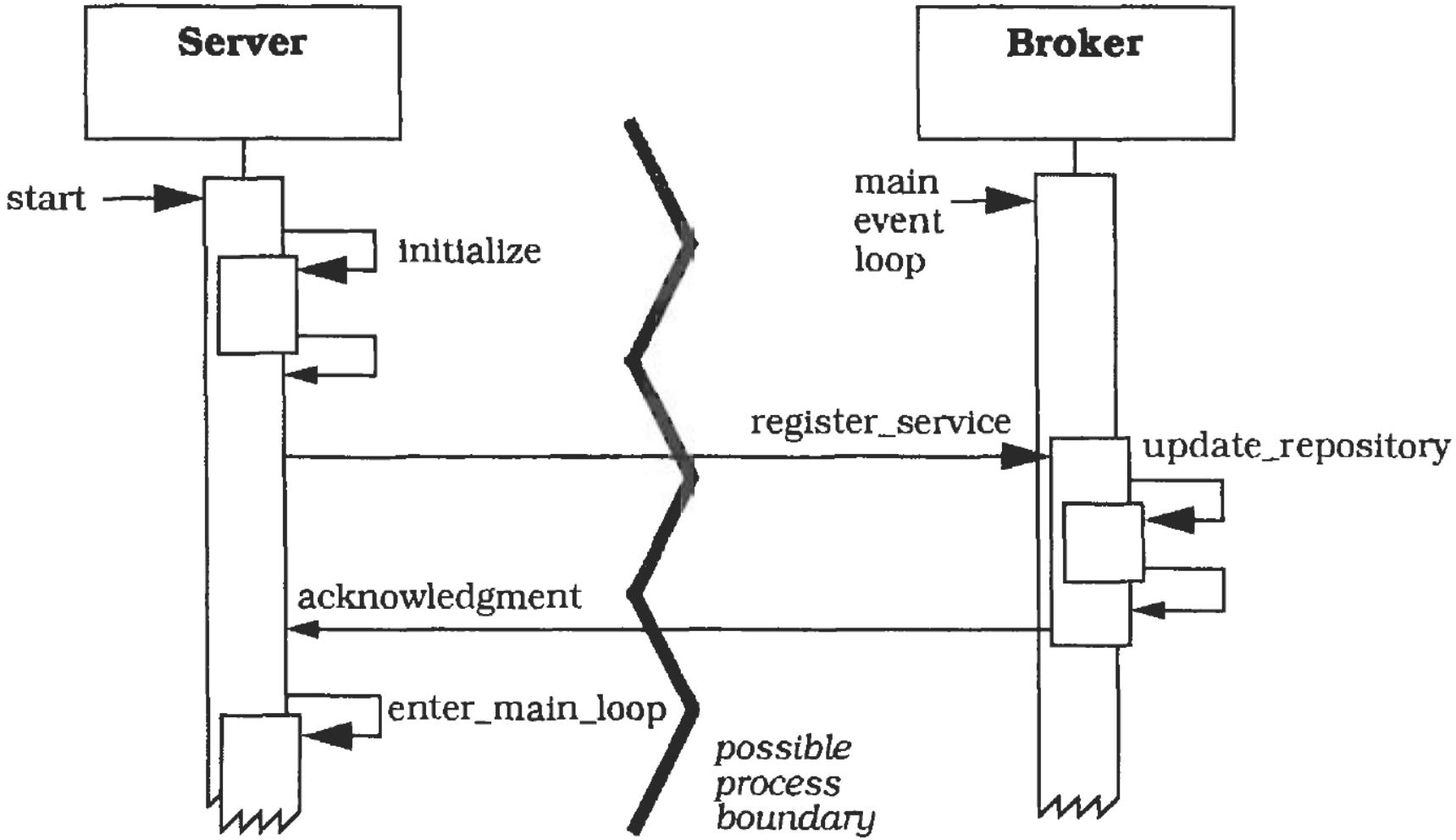
- **Server-side Proxy**
  - 특정 시스템에 맞춰진 기능을 캡슐화, 서버 내에 있는 서비스를 호출, 서버와 브로커 사이를 중재
  - **Client Proxy**와의 차이점
    - 요청 들어온 메시지를 원래대로 풀고**(unpack)**, 매개변수를 언마샬링**(unmarshalling)**하여 적절한 서비스를 호출
  - 결과와 예외를 클라이언트에 보내기 전, 마샬링을 수행하기 위해 사용
- **Bridge**
  - 특정 네트워크에 맞춰진 기능을 캡슐화
  - 원격 브로커의 브리지와 로컬 브로커 사이를 중재

Broker Pattern: Structure

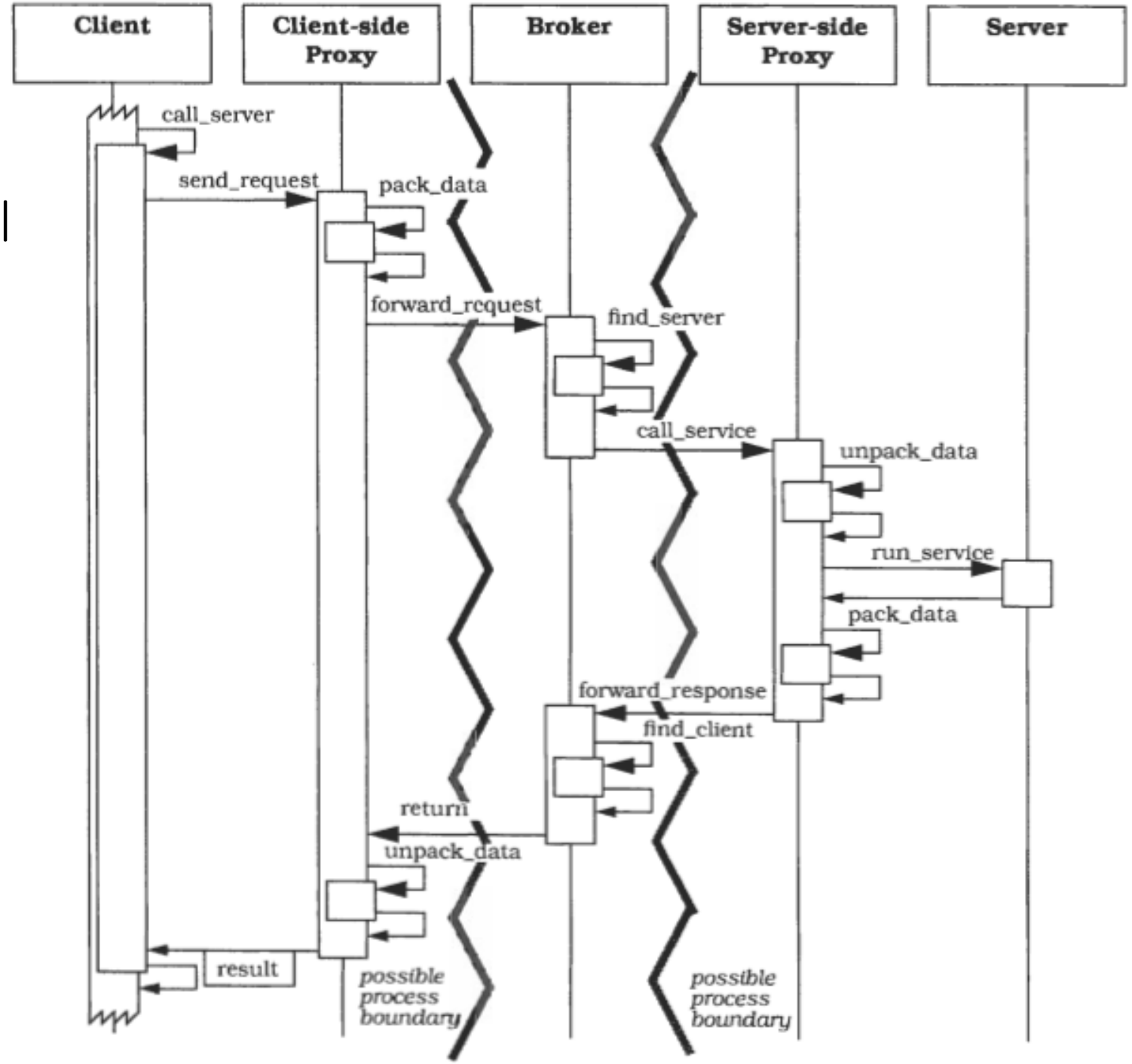


Broker Pattern: Behavior

- 서버가 자신을 로컬 브로커 컴포넌트에 등록

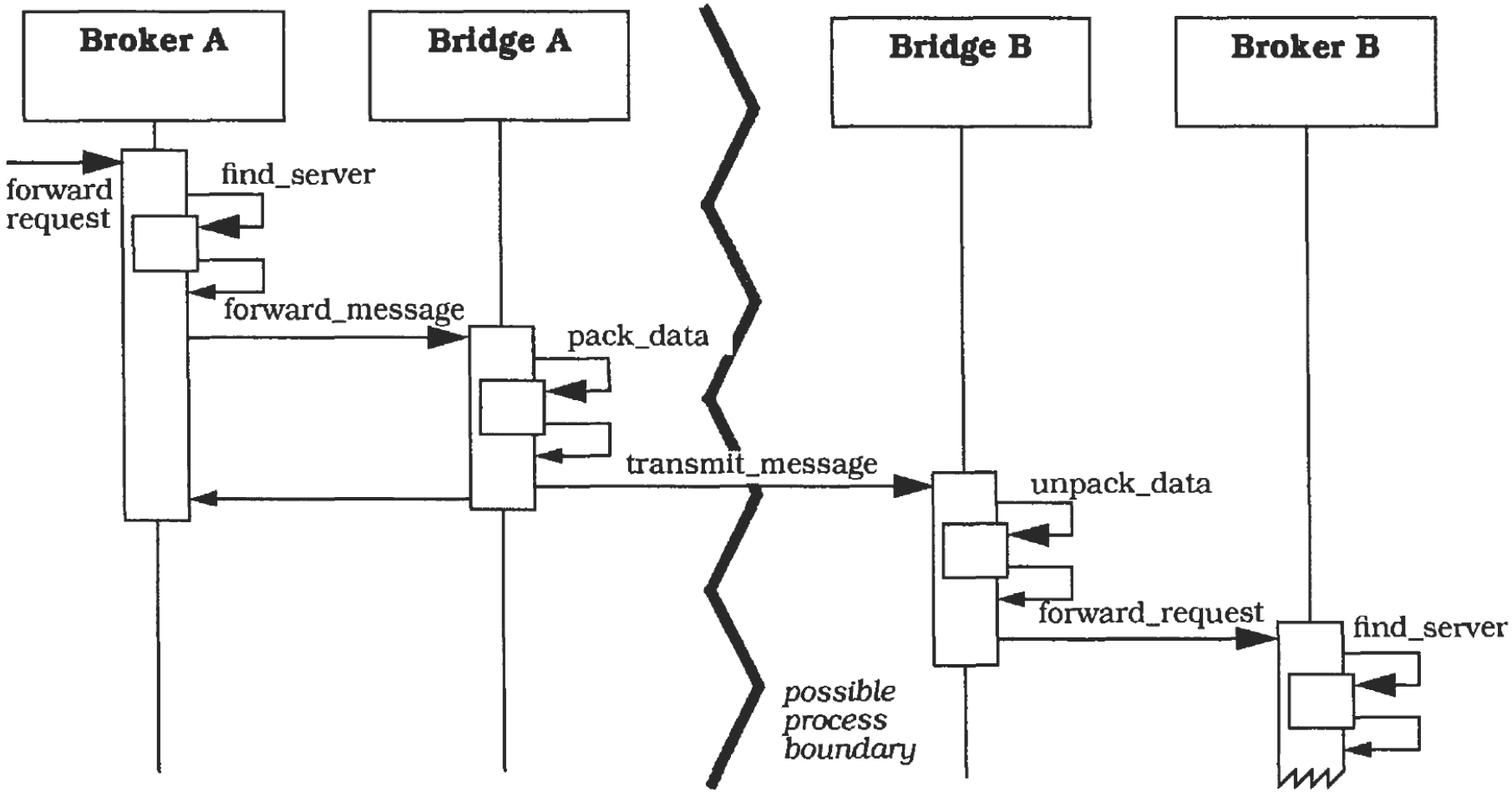


- 클라이언트가 로컬 서버에 요청을 보낼 때



Broker Pattern: Behavior

- 브로커의 상호작용



Broker Pattern: Realization

- 설계 순서
  1. 객체 모델을 정의하거나 기존 모델을 재사용 할 지 결정
     - 객체 이름, 객체
     - 요청, 값, 예외
     - 지원 타입
     - 타입 확장
     - 인터페이스
     - 오퍼레이션 등 정의
     - 기반 계산 모델**(underlying computational model)**을 서술하는 것이 중요
  2. 컴포넌트들 사이의 상호 연동을 어떤 방식으로 할 지 결정
     - 바이너리 표준 혹은 상위레벨 **IDL(Interface Definition Language)**를 도입

Broker Pattern: Realization

- 설계 순서
  3. 클라이언트와 서버 간의 협력을 위한 **Broker** 컴포넌트가 제공할 **API** 정의
     - 클라이언트의 경우 생성한 요청을 브로커에게 넘기고 그에 대한 응답을 받은 기능 사용
     - 서버의 경우 등록하기 위해 **API** 사용
     - (서버의 경우) 또한 적절한 서버 식별 메커니즘도 필요
  4. **Proxy** 객체를 사용해 환경과 관련된 부분 캡슐화 설계
     - **Client Proxy**는 클라이언트에게는 서버로, **Server Proxy**는 서버에게는 클라이언트로 간주

Broker Pattern: Realization

- 설계 순서
  5. **Broker** 컴포넌트 설계
     1. **Client Proxy**와 **Server Proxy**의 상호작용을 위한 프로토콜 세부 정의
     2. 로컬 브로커가 네크워크의 모든 머신이 사용할 수 있도록 프로토콜 세부 정의
     3. 호출을 보낸 클라이언트에게 모든 결과와 예외를 다시 반환하기 위한 클라이언트 기억
     4. 프록시가 매개변수와 결과에 대한 데이터 팩/언팩의 기능을 제공하지 않으면 해당 기능 포함
     5. 비동기 통신을 지원할 경우, 이를 위한 메시지 버퍼 제공
     6. 서버의 물리적 위치를 기억하기 위한 디렉토리 서비스 갖춤
     7. 서버 고유의 식별자를 동적으로 생성하기 위한 네임 서비스 제공
     8. 동적 메서드 호출을 지원하기 위하여 기존 서버의 타임 정보 저장 기능 구현
     9. 실패할 경우 고려, 통신 실패 등의 경우 어떻게 오류를 핸들링할 것인지 정의
  6. **IDL** 컴파일러 설계

Broker Pattern: Implementation

- Server Code

```
...
// Create an object request broker
ORB orb = ORB.init(args, null);

// Create a new address book ...
AddressBookServant servant = new AddressBookServant();

// ... and connect it to our orb
orb.connect(servant);

// Obtain reference for our nameservice
org.omg.CORBA.Object object = orb.resolve_initial_references("NameService");

// Since we have only an object reference, we must cast it to a NamingContext. We use a helper class for this purpose
NamingContext namingContext = NamingContextHelper.narrow(object);

// Add a new naming component for our interface
NameComponent list[] = { new NameComponent("address_book", "") };

// Now notify naming service of our new interface
namingContext.rebind(list, servant);
```

Broker Pattern: Implementation

- Client Code

```
// import servant class that is generated by IDL  converter
import  address_book_system.address_bookPackage.*;

...
// Create an object request broker
ORB orb = ORB.init(args, null);

// Obtain object reference for name service ...
org.omg.CORBA.Object object =
orb.resolve_initial_references("NameService");

// ... and narrow it to a NameContext
NamingContext namingContext =
NamingContextHelper.narrow(object);

// Create a name component array
NameComponent nc_array[] =
{ new NameComponent("address_book","") };
```

```
// Get an address book object reference ...
org.omg.CORBA.Object objectReference =
namingContext.resolve(nc_array);

// ... and narrow it to get an address book
address_book AddressBook =
address_bookHelper.narrow(objectReference);

// call the  address book interface
name = AddressBook.name_from_email(email);
```

Broker Pattern: Implementation

- EJB3.0 Code

- Server

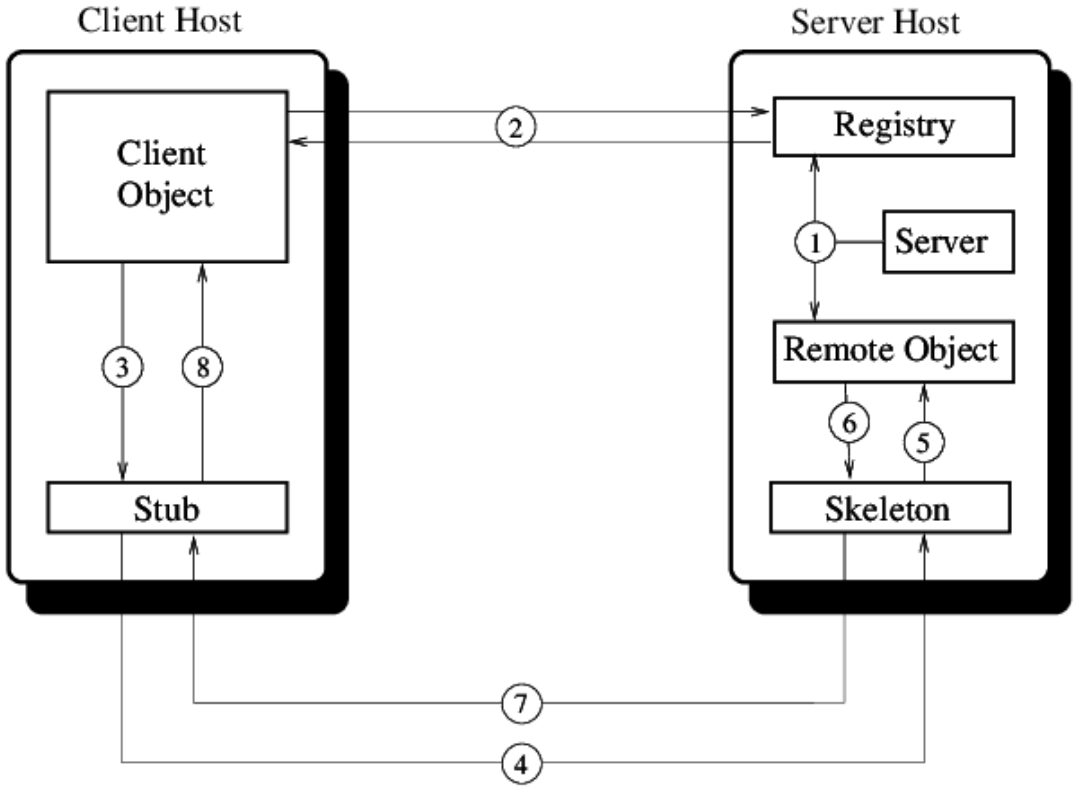
```
…
@Stateless(name=“HelloBean")
@Remote(HelloRemote.class)

public class HelloWorldBean {
        public String sayHello() {
                return "Hello World!!!";
        }
}
```

- Client

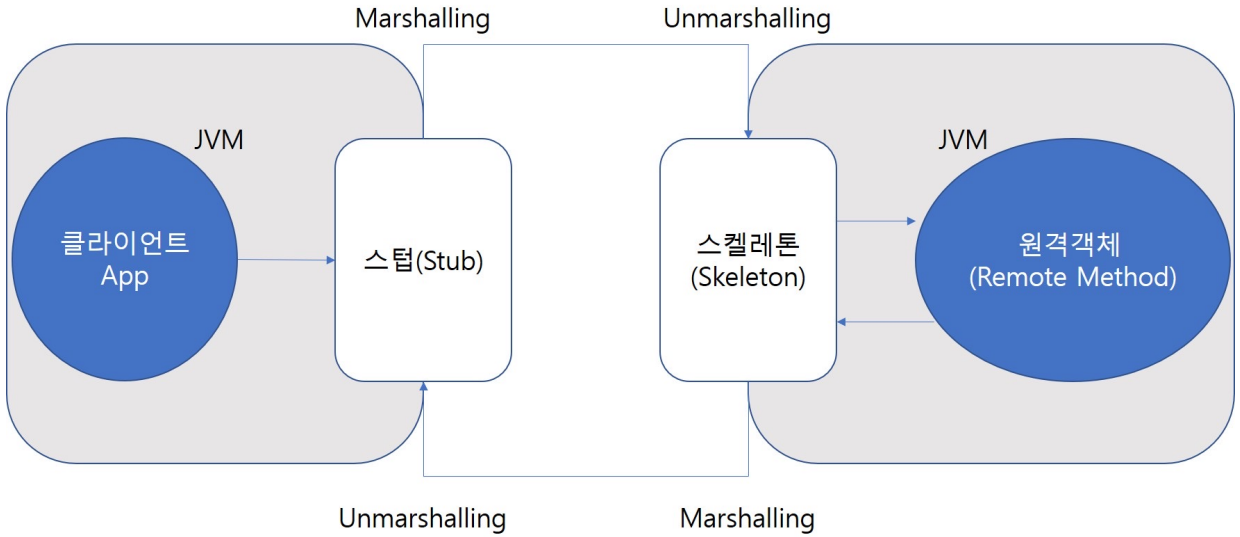
```
…
InitialContext ic = new InitialContext();
Hello = (HelloRemote) ic.lookup("example/HelloBean/remote");
…
```

Broker Pattern: Case Studies

- **RMI 내부**

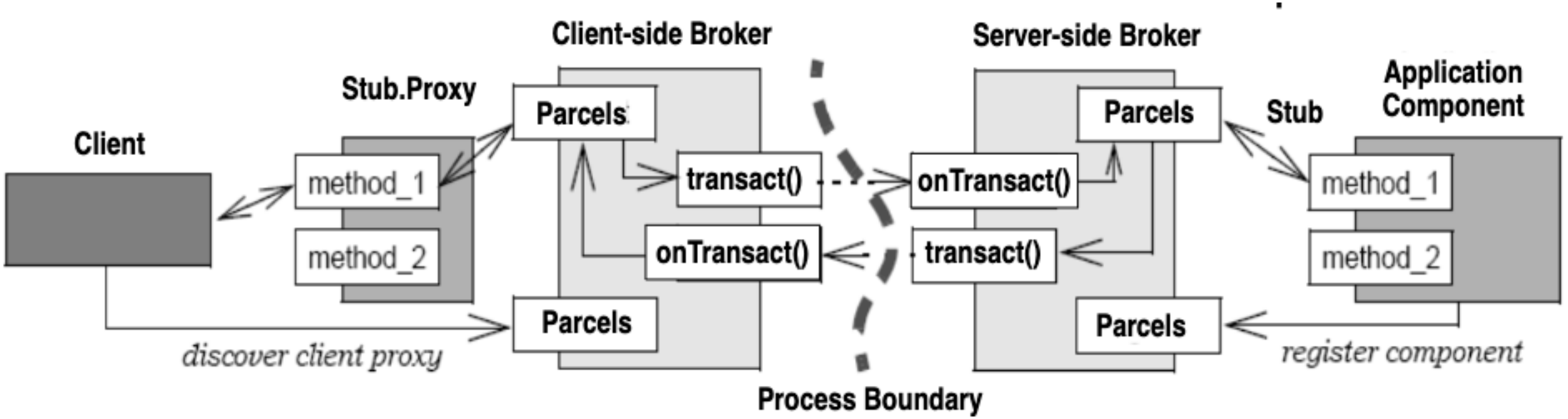


Java RMI architecture.



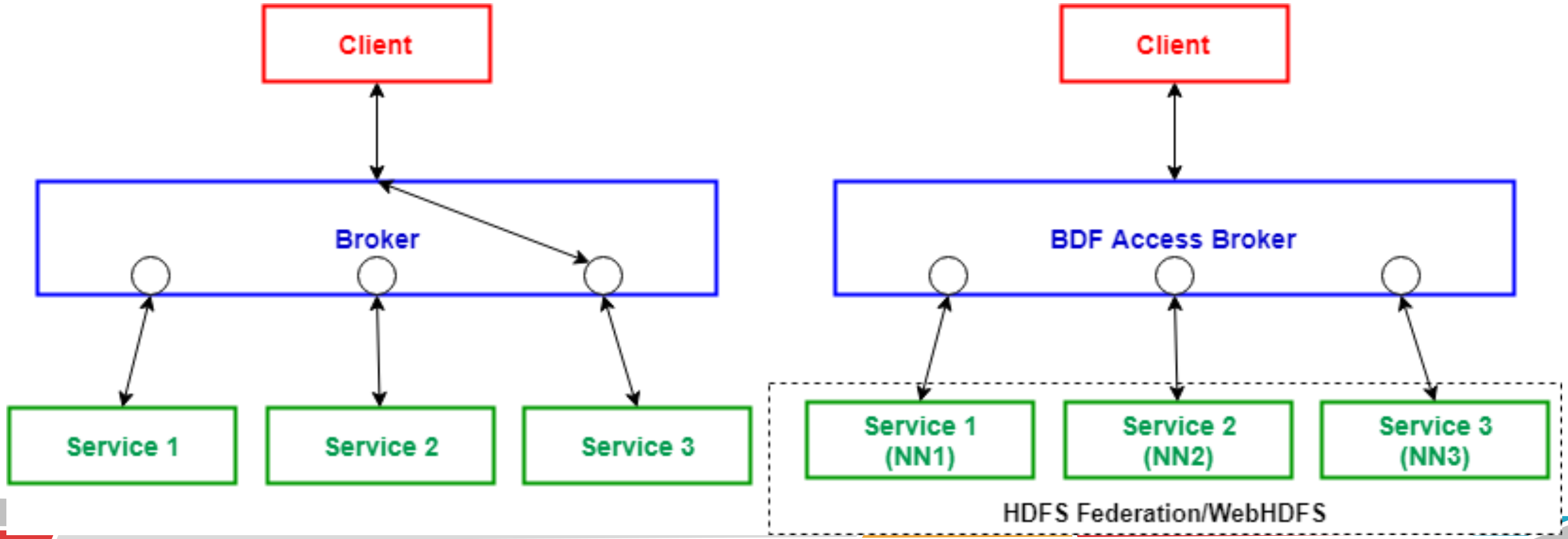
Broker Pattern: Case Studies

- Android Bound Services uses Broker to invoke methods across processes



Broker Pattern: Case Studies

- A high-level abstraction of general access broker pattern vs. the proposed BDF access broker



Broker Pattern: Benefits

- 컴포넌트간의 **위치 투명성(Location Transparencey)**을 제공
- 컴포넌트의 **가변성(Changeability)**과 **확장성(Extensibility)**이 보장
- 플랫폼 간의 **이식가능성(Portability)** 제공함
- 서로 다른 **Broker** 시스템들 간의 상호 운영성을 지원
- 재사용 컴포넌트 확보에 용이**(Reusability)**

Broker Pattern: Liabilities

- 성능에 대한 불이익
- 브로커 장애 발생 시를 대비해, 컴포넌트를 복제해 둠으로써 신뢰성을 보장해야 함
- 테스트 디버깅의 복잡함 (서버, 클라이언트 연동 시에)
- 분산 환경을 지원하는 시스템이 많지 않음
  - **CORBA, DCOM, EJB**는 이전 버전
  - **Apache Kafka, Jboss Messaging**과 같은 메시지 브로커가 최근 활용**(Pub-Sub** 패턴**)**

Question?

Seonah Lee

saleese@gmail.com