Oracle Net Services 및 연결 아키텍처 — 기술 문서

1. How Oracle Net Services Works — 아키텍처적 분석

1.1 왜 이 메커니즘이 필요한가 (Why)

Oracle Net Services의 존재 이유를 이해하려면, 애플리케이션과 데이터베이스 사이의 통신 문제를 먼저 파악해야 한다.

클라이언트 애플리케이션(Spring Boot)이 Oracle Database에 SQL을 전달하려면 두 가지 근본적인 문제를 해결해야 한다. 첫째, 네트워크 프로토콜의 이질성이다. TCP/IP, DECnet, LU6.2 등 다양한 네트워크 프로토콜이 존재하며, 애플리케이션이 이들 각각에 직접 대응하는 것은 비효율적이다. 둘째, 데이터 표현의 이질성이다. 클라이언트와 서버가 서로 다른 OS, 다른 endianness, 다른 문자 인코딩을 사용할 수 있다.

Oracle Net Services는 이 두 문제를 추상화 계층으로 해결한다. 애플리케이션은 Oracle Net Services API만 호출하면 되고, 실제 네트워크 통신의 복잡성은 Oracle Net Services가 처리한다.

1.2 동작 메커니즘 (How)

공식 문서의 "How Oracle Net Services Works" 섹션은 핵심만 간결하게 기술한다:

> Oracle Database protocols accept SQL statements from the interface of the Oracle applications, and then package them for transmission to Oracle Database. Transmission occurs through a supported industry-standard higher level protocol or API. Replies from Oracle Database are packaged through the same higher level communications mechanism. This work occurs independently of the network operating system.

이 설명을 구체적인 아키텍처로 풀어내면 다음과 같다.

1단계: 클라이언트 측 패키징

JDBC 드라이버(또는 OCI)가 애플리케이션의 SQL 문장을 받으면, Oracle Net Services의 클라이언트 측 구성요소가 이를 **Oracle Net 프로토콜 데이터 단위(Protocol Data Unit)**로 패키징한다. 이 패키징 과정에서 SQL 텍스트, 바인드 변수 값, 메타데이터가 Oracle이 정의한 포맷으로 직렬화된다. 중요한 점은 이 포맷이 네트워크 프로토콜(TCP/IP든 다른 것이든)과 독립적이라는 것이다.

2단계: 네트워크 전송

패키징된 데이터는 Oracle Net Services의 드라이버 계층을 통해 실제 네트워크 프로토콜로 전송된다. 현대 환경에서는 거의 대부분 TCP/IP를 사용한다. 드라이버 계층이 Oracle Net 프로토콜 데이터를 TCP 세그먼트로 분할하고 전송한다.

3단계: 서버 측 수신 및 처리

서버 측에서는 Listener가 초기 연결 요청을 받고, 연결이 수립된 후에는 Server Process가 직접 클라이언트와 통신한다. Server Process는 수신된 패킷을 언패킹하여 SQL 문장을 추출하고, 이를 실행한 후 결과를 다시 패키징하여 반환한다.

이 과정을 도식화하면:

```
[Application (SQL*Plus, JDBC)]
         |
         ▼
[Oracle Call Interface / JDBC Thin Driver]
         |  SQL 문장 + 바인드 변수
         ▼
[Oracle Net Services - Client Side]
         |  패키징 (Protocol Data Unit 생성)
         ▼
[Network Driver (TCP/IP)]
         |  TCP 세그먼트화 및 전송
         ▼
   ═══════════════════════════════════════
        Network (TCP/IP)
   ═══════════════════════════════════════
         |
         ▼
[Oracle Net Services - Server Side]
         |  언패킹
         ▼
[Server Process]
         |  SQL 실행 → SGA 접근 → 결과 생성
         ▼
[Oracle Net Services - Server Side]
         |  결과 패키징
         ▼
   (역방향으로 전송)
```

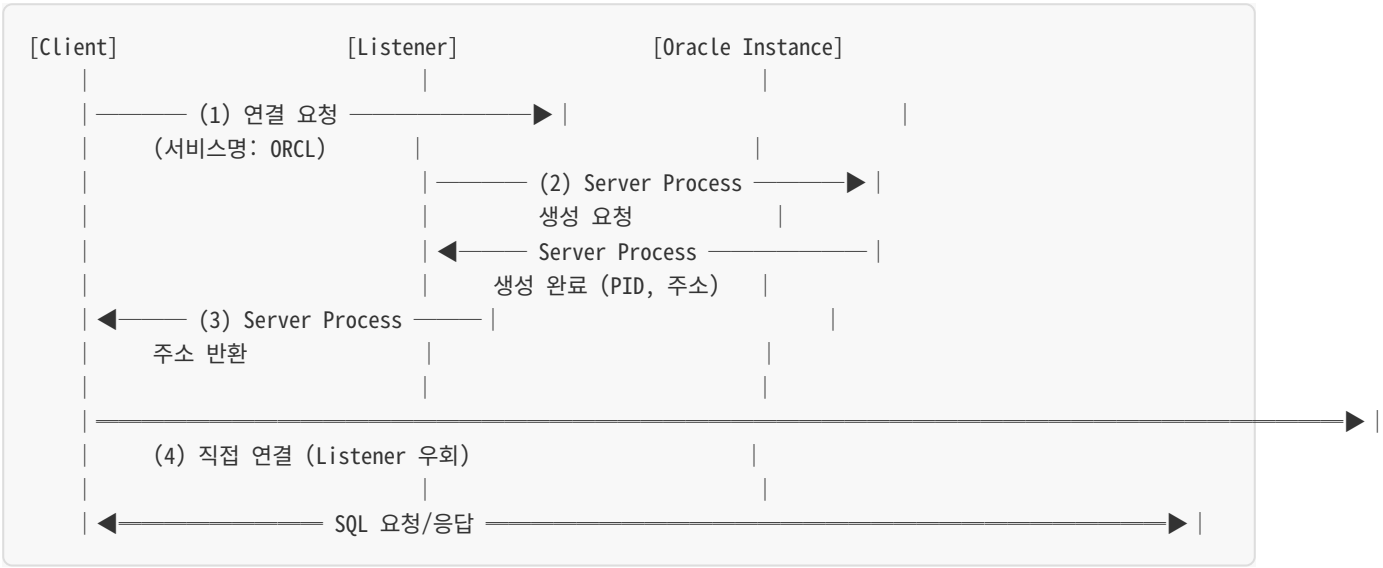
1.3 Listener의 역할과 Connection 수립 과정

"How Oracle Net Services Works"를 이해하려면 Listener의 동작을 반드시 알아야 한다. Listener는 연결 수립 시점에만 관여하며, 연결이 수립된 후에는 통신에서 빠진다.

연결 수립 과정은 다음과 같다:

1. 클라이언트가 Listener에 연결 요청을 보낸다. 이 요청에는 서비스 이름(예: ORCL)이 포함된다.
2. Listener가 Service Handler를 선택한다. Dedicated Server 모드에서는 Listener가 새로운 Server Process 생성을 유발하고, 그 프로세스의 주소를 클라이언트에게 반환한다.
3. 클라이언트가 Server Process에 직접 연결한다. 이 시점부터 Listener는 더 이상 관여하지 않는다.
4. Server Process가 PGA를 할당하고 세션을 초기화한다.

이 과정에서 발생하는 비용이 "Connection 생성 비용"의 실체다.



1.4 Service Registration 메커니즘

Listener가 어떤 Instance에 어떤 서비스가 있는지 아는 방법은 Service Registration이다. Instance 내의 LREG (Listener Registration) 프로세스가 주기적으로 Listener에게 다음 정보를 등록한다:

- 데이터베이스 서비스 이름
- Instance 이름
- 현재 로드 (CPU 사용률, 연결 수)
- 사용 가능한 Service Handler 목록

이 동적 등록 덕분에 listener.ora 파일에 모든 Instance를 정적으로 설정할 필요가 없다. Instance가 시작되면 자동으로 Listener에 등록되고, 종료되면 등록이 해제된다.

1.5 Dedicated Server vs Shared Server

Oracle Net Services는 두 가지 연결 아키텍처를 지원한다.

Dedicated Server (기본값)는 클라이언트 하나당 Server Process 하나를 할당한다. 클라이언트가 유휴 상태여도 Server Process는 유지되며, 이것이 리소스 낭비의 원인이 된다.

Shared Server는 Dispatcher가 여러 클라이언트의 요청을 받아 공유 Server Process 풀에 분배한다. 요청 단위로 Server Process를 할당하므로 리소스 효율이 높다. 그러나 요청 큐잉 오버헤드가 있고, UGA가 SGA(Large Pool이 구성되면 Large Pool, 아니면 Shared Pool)에 저장되어 SGA 크기가 증가한다.

Wherehouse 프로젝트처럼 Spring Boot + HikariCP 환경에서는 Dedicated Server 모드가 일반적이다. HikariCP가 Connection Pool을 관리하고, 각 Connection은 Oracle의 Dedicated Server Process에 1:1로 매핑된다.

1.6 백엔드 개발자 관점의 설계적 의의 (So What)

"How Oracle Net Services Works"가 Wherehouse 프로젝트의 성능 분석에 주는 시사점은 다음과 같다.

Connection 생성 비용의 구체적 내역: 공식 문서에서 "드라이버 소프트웨어를 로드하고 추가 백그라운드 프로세스를 시작할 수 있다"고 언급한 부분은, Dedicated Server 모드에서 새로운 Server Process가 생성되는 과정을 지칭한다. 이 Server Process 생성 + PGA 할당 비용이 Connection Pool 없이 매번 연결을 생성하면 안 되는 아키텍처적 근거다.

HikariCP Waiting의 의미 재해석: 1차 테스트에서 HikariCP Waiting이 최대 9건 발생했다. 이는 Oracle Net Services 레벨에서 보면, **이미 수립된 Session(Server Process가 할당된 연결)**을 애플리케이션 스레드들이 경쟁적으로 획득하려는 상황이다. N+1로 25번 쿼리가 발생할 때, 각 쿼리마다 Pool에서 Connection을 획득 → SQL 전송(Oracle Net Services 경유) → 결과 수신 → Connection 반환 사이클이 반복된다. Pool 크기가 10개이므로, 동시 요청이 10개를 초과하면 대기가 발생한다.

네트워크 왕복(Round-Trip)의 비용: Oracle Net Services는 매 SQL 호출마다 패키징 → 전송 → 언패킹 과정을 거친다. N+1 문제에서 25번의 SELECT가 발생하면 최소 25번의 네트워크 왕복이 발생한다. Bulk Fetch나 Chunk 방식은 이 왕복 횟수를 줄여 Oracle Net Services 계층의 오버헤드를 감소시킨다.

1.7 정리

"How Oracle Net Services Works"의 핵심은 추상화와 독립성이다. 애플리케이션 레벨의 SQL 요청을 네트워크 프로토콜과 OS로부터 분리하여, 어떤 환경에서든 동일한 방식으로 Oracle Database와 통신할 수 있게 한다. 백엔드 개발자 입장에서 이 계층의 존재를 인식해야 하는 이유는, Connection 생성 비용, 네트워크 왕복 비용, Connection Pool의 필요성이 모두 이 아키텍처에서 기인하기 때문이다.

---

2. Service Handler — 아키텍처적 분석

2.1 정의와 존재 이유 (Why)

공식 문서의 정의를 먼저 보자:

> A service handler is a dedicated server process or dispatcher that acts as a connection point to a database.
> During registration, the LREG process provides the listener with the instance name, database service names, and the type and addresses of service handlers.

Service Handler는 클라이언트가 데이터베이스에 접근하는 실제 진입점이다. Listener는 연결 요청을 받지만, 실제 SQL을 처리하지는 않는다. Listener의 역할은 적절한 Service Handler를 선택하여 클라이언트를 연결해주는 것이다.

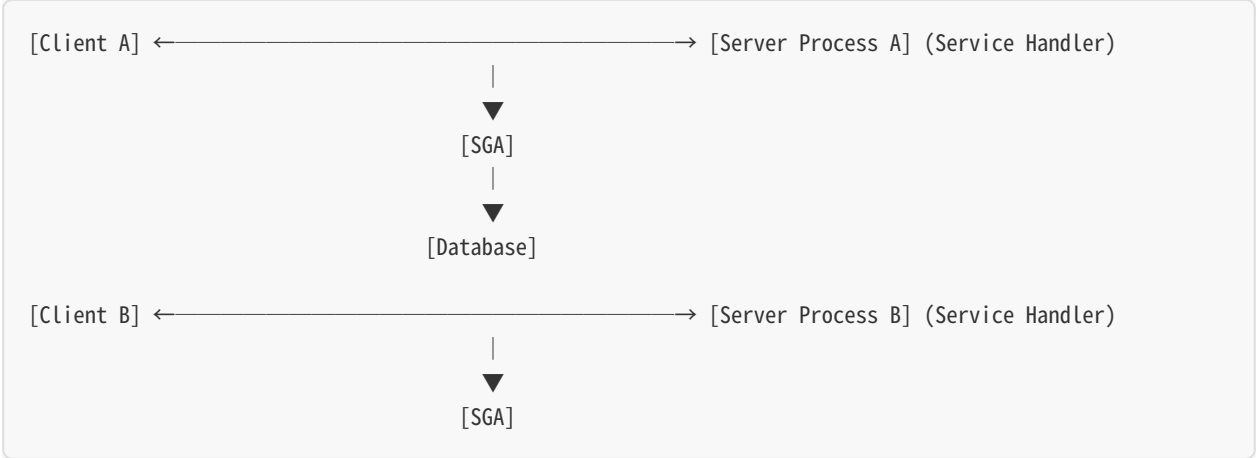
Service Handler가 필요한 이유는 관심사의 분리 때문이다. Listener가 모든 클라이언트의 SQL 요청을 직접 처리한다면, Listener는 단일 장애점(Single Point of Failure)이 되고 확장이 불가능하다. 대신 Listener는 연결 수립만 담당하고, 실제 작업은 Service Handler에게 위임한다.

2.2 Service Handler의 두 가지 유형 (How)

Service Handler는 연결 아키텍처에 따라 두 가지 형태로 존재한다.

2.2.1 Dedicated Server Process (전용 서버 프로세스)

Dedicated Server 모드에서 Service Handler는 Server Process 자체다. 클라이언트 연결 요청이 들어오면 Listener가 새로운 Server Process 생성을 유발하고, 이 프로세스가 해당 클라이언트 전용 Service Handler가 된다.

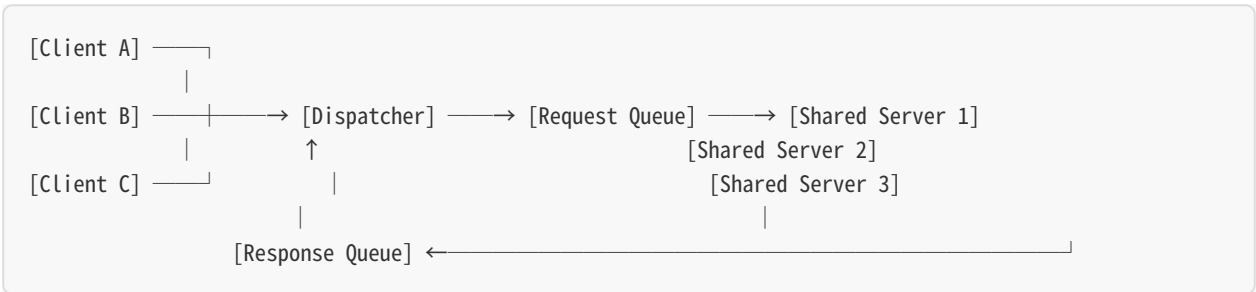


이 모드에서 Service Handler의 특징은 다음과 같다:

- 클라이언트와 1:1 관계를 갖는다
- 클라이언트가 연결을 끊을 때까지 해당 클라이언트만 서비스한다
- 각 Service Handler(Server Process)는 자신만의 PGA를 갖는다
- 클라이언트가 유휴 상태여도 프로세스는 유지된다

2.2.2 Dispatcher (공유 서버 모드)

Shared Server 모드에서 Service Handler는 Dispatcher다. Dispatcher는 여러 클라이언트의 요청을 받아 공유 Server Process 풀에 분배한다.



이 모드에서 Service Handler(Dispatcher)의 특징:

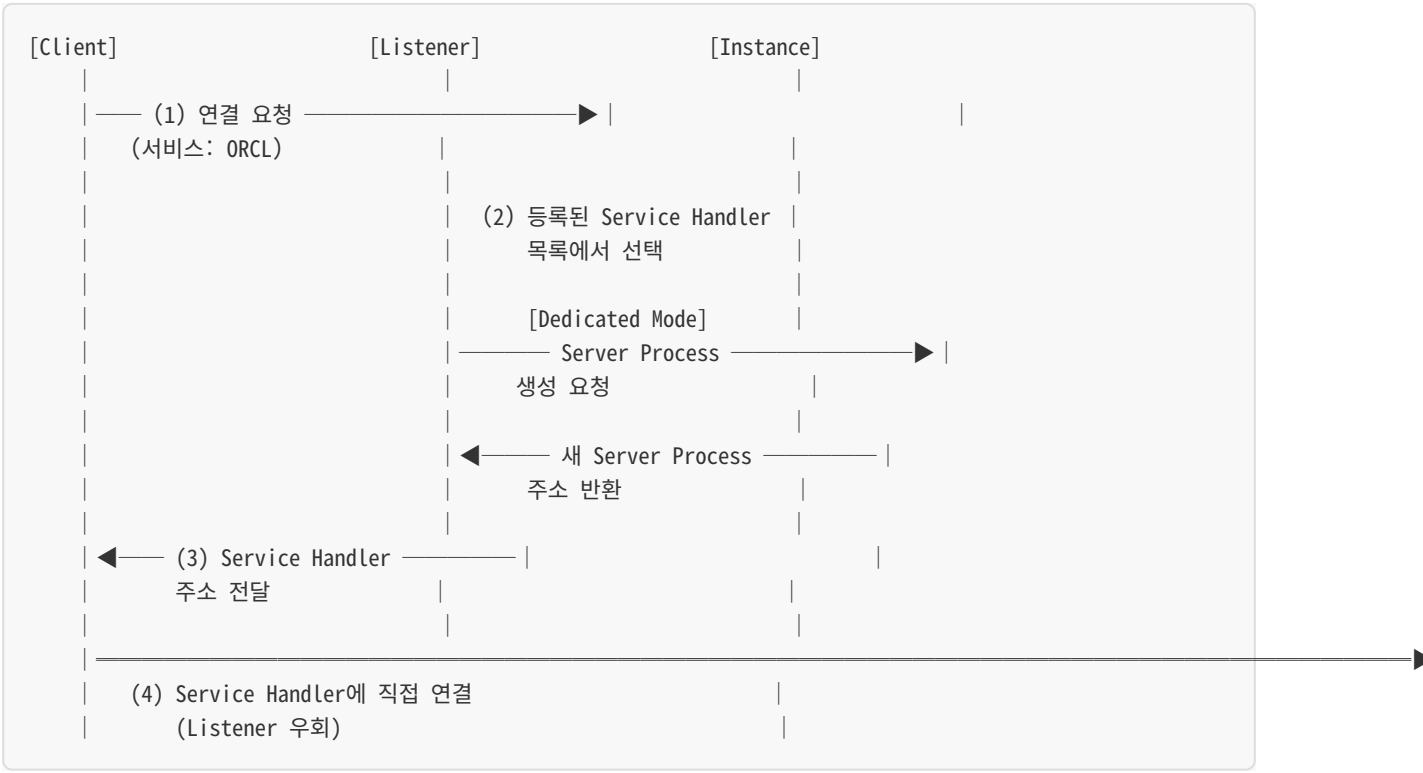
- 여러 클라이언트와 N:1 관계를 갖는다
- 요청을 큐에 넣고 가용한 Shared Server가 처리한다
- UGA가 SGA(Large Pool 또는 Shared Pool)에 저장된다 (PGA가 아님)

- 리소스 효율은 높지만 요청 큐잉 오버헤드가 존재한다

2.3 Listener와 Service Handler의 상호작용

Listener가 Service Handler를 인식하는 과정은 Service Registration을 통해 이루어진다. LREG 프로세스가 Listener에게 등록하는 정보에는 "사용 가능한 Service Handler의 유형과 주소"가 포함된다.

연결 수립 시 Listener의 동작을 상세히 보면:



핵심은 (3)번 단계다. Listener는 클라이언트에게 Service Handler의 주소를 반환하고, 이후 클라이언트는 Listener를 거치지 않고 Service Handler와 직접 통신한다.

2.4 백엔드 개발자 관점의 설계적 의의 (So What)

2.4.1 HikariCP Connection의 실체

Spring Boot + HikariCP 환경에서 "Connection"이라고 부르는 것의 Oracle 측 실체가 바로 Service Handler(Dedicated Server Process)다. HikariCP Pool에 10개의 Connection이 있다면, Oracle Instance에는 해당 애플리케이션을 위한 10개의 Server Process가 존재한다.

2.4.2 Connection 생성 비용의 구체적 내역

"Connection 생성이 비싸다"는 말의 아키텍처적 의미:

1. Listener가 OS에 Server Process fork 요청
2. OS가 새 프로세스 생성 (메모리 공간 할당, 프로세스 테이블 등록)
3. Server Process가 PGA 메모리 할당
4. Server Process가 SGA에 접근 권한 획득
5. 세션 정보 초기화

이 모든 과정이 Service Handler(Server Process) 생성 과정이다. Connection Pool을 사용하면 이 과정을 애플리케이션 시작 시 한 번만 수행하고, 이후에는 이미 생성된 Service Handler를 재사용한다.

2.4.3 Wherehouse 프로젝트 1차 테스트 데이터 재해석

1차 테스트에서 RDB 시간 비율이 6.2%인데 전체 응답이 5,531ms인 상황을 Service Handler 관점에서 분석하면:

지표	값	아키텍처적 의미
HikariCP Pool Size	10	Oracle 측 Service Handler 10개 유지
RDB 조회 시간	343ms	Server Process의 실제 작업 시간
RDB 시간 비율	6.2%	Server Process 활용률
총 소요 시간	5,531ms	나머지 93.8%는 어디서 소요?
HikariCP Waiting	최대 9건	Connection(Server Process) 획득 경합

중요한 점은 Service Handler 자체의 처리 시간(RDB 조회 343ms)은 짧았다는 것이다. 병목은 Service Handler에서 발생한 것이 아니라, 애플리케이션 레벨에서 Service Handler(Connection)를 획득하기 위해 대기한 시간이 누적된 것이다.

2.4.4 Service Handler 수와 동시성의 관계

Service Handler 개수(= Connection Pool 크기)는 애플리케이션의 동시 처리 능력을 결정한다. 그러나 무조건 늘리는 것이 답은 아니다:

- Service Handler가 많아지면 → PGA 메모리 총량 증가 → 서버 메모리 부담
- Service Handler가 많아지면 → Context Switching 오버헤드 증가
- Service Handler가 적으면 → Connection 경합 증가 → 대기 시간 증가

Wherehouse 프로젝트의 경우, N+1 문제를 해결하여 쿼리 횟수를 줄이는 것이 Service Handler(Connection) 개수를 늘리는 것보다 근본적인 해결책이었다. 3차 테스트(Chunk 방식)에서 RDB 호출 횟수가 9회로 줄면서 전체 응답 시간이 4,434ms로 개선된 것이 이를 증명한다.

2.5 정리

Service Handler는 클라이언트가 Oracle Database와 실제로 통신하는 엔드포인트다. Dedicated Server 모드에서는 Server Process가, Shared Server 모드에서는 Dispatcher가 Service Handler 역할을 한다. 백엔드 개발자가 다루는 "Connection"의 Oracle 측 실체가 이 Service Handler이며, Connection Pool 크기 = 할당된 Service Handler 개수 = 동시 SQL 처리 능력이라는 등식이 성립한다.

---

3. "The Oracle Net Listener" 섹션 추가 학습 필요성 분석

해당 섹션의 하위 구조를 보면:

```
The Oracle Net Listener
 ├─── Service Names
 ├─── Services in a Multitenant Environment
 │    ├─── Service Creation (Default / Nondefault Services)
 │    └─── Connections to Containers in a CDB
 └─── Service Registration
```

3.1 학습 권장: Service Names

권장 이유: JDBC Connection String 작성과 직접 연관된다.

공식 문서의 핵심 내용:

> A service name is a logical representation of a service used for client connections. When a client connects to a listener, it requests a connection to a service. When a database instance starts, it registers itself with a listener as providing one or more services by name.
> A single service, as known by a listener, can identify one or more database instances. Also, a single database instance can register one or more services with a listener.

이 개념이 중요한 이유는, Spring Boot의 application.yml에서 작성하는 Connection String이 이 Service Name을 기반으로 동작하기 때문이다:

```
spring:
  datasource:
    url: jdbc:oracle:thin:@//hostname:1521/ORCL  # ← ORCL이 Service Name
```

Service Name의 설계적 의의:

구조	설명	실무적 의미
1 Service → N Instance	하나의 서비스가 여러 인스턴스를 가리킴	RAC 환경에서 로드 밸런싱, Failover
1 Instance → N Service	하나의 인스턴스가 여러 서비스로 등록	동일 DB를 용도별로 구분 (예: 읽기 전용 vs 쓰기)

Wherehouse 프로젝트처럼 단일 인스턴스 환경에서는 복잡하게 생각할 필요 없지만, 면접에서 "Service Name이 뭔가요?"라는 질문에 "Listener가 클라이언트 요청을 적절한 Instance로 라우팅하기 위한 논리적 식별자"라고 답할 수 있으면 충분하다.

3.2 학습 불필요: Services in a Multitenant Environment

스킵 권장 이유: CDB/PDB 아키텍처는 DBA 영역이다.

Oracle 12c 이후 도입된 Multitenant Architecture(CDB: Container Database, PDB: Pluggable Database)는 하나의 물리적 데이터베이스 안에 여러 논리적 데이터베이스를 운영하는 구조다. 클라우드 서비스나 대규모 엔터프라이즈 환경에서 사용되지만, 백엔드 개발자가 직접 다룰 일은 거의 없다.

애플리케이션 입장에서는 PDB든 일반 DB든 Connection String만 올바르게 지정하면 동일하게 동작한다. 내부 구조를 알 필요가 없다.

3.3 이미 충분: Service Registration

앞서 설명한 내용으로 백엔드 개발자 수준에서는 충분하다:

- LREG 프로세스가 Listener에게 Instance 정보를 동적으로 등록
- 등록 정보: 서비스 이름, Instance 이름, Service Handler 목록, 현재 로드
- listener.ora에 정적 설정 없이도 Instance가 자동으로 Listener에 등록됨

이 이상의 상세(등록 주기, 파라미터 튜닝 등)는 DBA 영역이다.

3.4 결론

섹션	학습 필요성	이유
Service Names	○ 권장	Connection String과 직접 연관, 면접 대비
Services in a Multitenant Environment	× 불필요	DBA 영역, 백엔드 개발자 목적 벗어남
Service Registration	△ 이미 충분	앞서 설명한 수준으로 충분

Service Names 섹션만 간략히 추가 학습하고, 이후 Dedicated Server Architecture 또는 Database Resident Connection Pooling (DRCP) 섹션으로 넘어가는 것이 Wherehouse 프로젝트 및 면접 목적에 부합한다.

---

4. Service Names — 아키텍처적 분석

4.1 정의와 존재 이유 (Why)

공식 문서의 정의:

> A service name is a logical representation of a service used for client connections. When a client connects to a listener, it requests a connection to a service. When a database instance starts, it registers itself with a listener as providing one or more services by name. Thus, the listener acts as a mediator between the client and instances and routes the connection request to the right place.

Service Name은 클라이언트가 데이터베이스에 연결할 때 사용하는 논리적 식별자다. 물리적인 호스트 주소나 Instance 이름이 아닌, 추상화된 서비스 단위로 연결을 요청한다.

이 추상화가 필요한 이유는 물리적 구조와 논리적 접근의 분리 때문이다. 클라이언트가 특정 호스트나 Instance에 직접 연결하면, 해당 서버가 장애를 일으키거나 구조가 변경될 때 애플리케이션을 수정해야 한다. Service Name을 통해 연결하면 물리적 구조가 바뀌어도 클라이언트 코드는 변경할 필요가 없다.

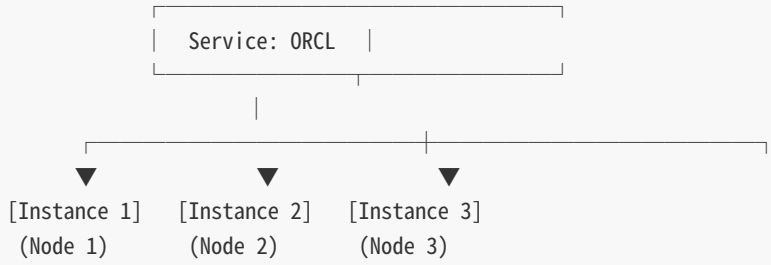
4.2 Service Name의 다대다 관계 (How)

공식 문서의 핵심 설명:

> A single service, as known by a listener, can identify one or more database instances. Also, a single database instance can register one or more services with a listener.

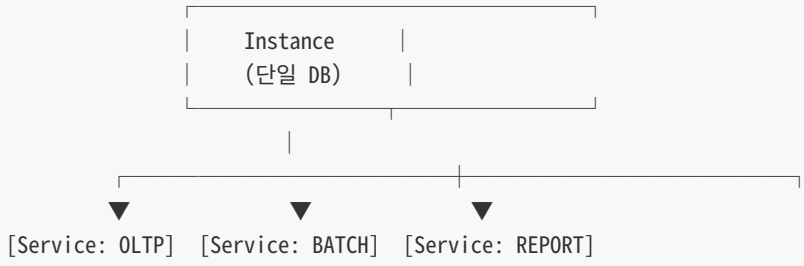
이 문장이 Service Name의 설계 철학을 담고 있다. Service Name과 Instance는 다대다(N:M) 관계다.

4.2.1 1 Service → N Instance (하나의 서비스가 여러 인스턴스를 가리킴)



Oracle RAC(Real Application Clusters) 환경에서 이 구조를 사용한다. 클라이언트는 "ORCL" 서비스에 연결을 요청하고, Listener가 현재 로드가 가장 적은 Instance로 라우팅한다. 특정 Instance가 장애를 일으키면 다른 Instance로 자동 Failover된다. 클라이언트는 물리적 구조를 알 필요가 없다.

4.2.2 1 Instance → N Service (하나의 인스턴스가 여러 서비스로 등록)



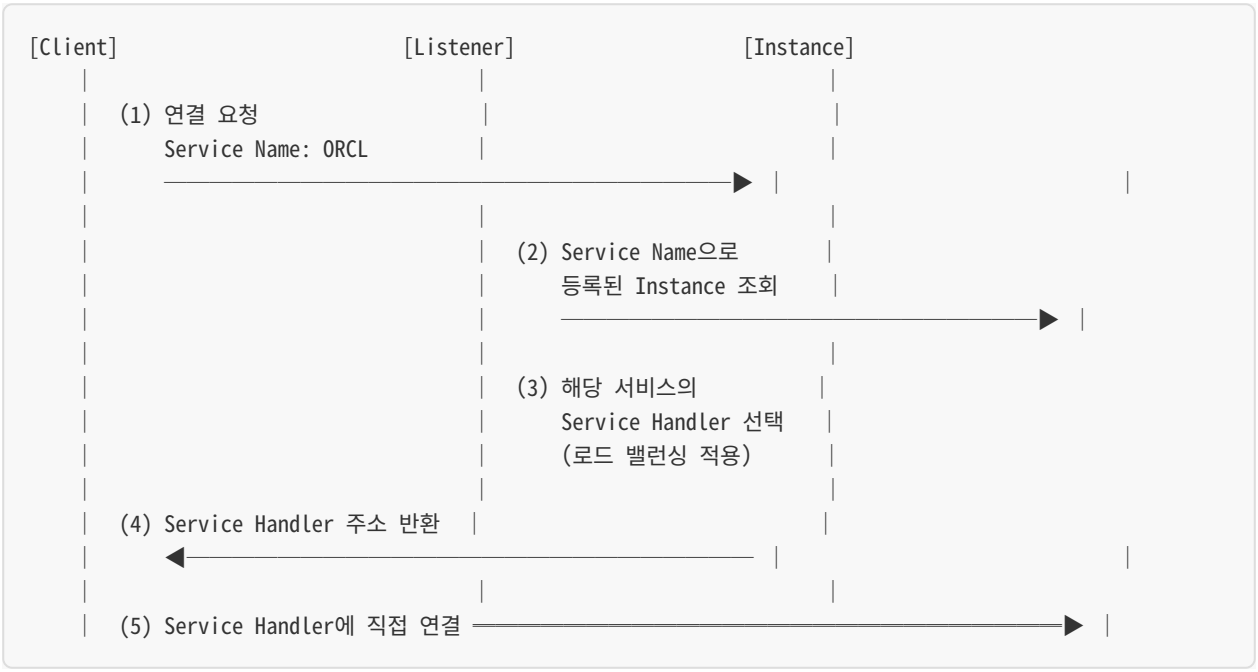
공식 문서의 예시가 이 경우다:

> Figure 18-6 shows one single-instance database associated with two services, book.example.com and soft.example.com. The services enable the same database to be identified differently by different clients. A database administrator can limit or reserve system resources, permitting better resource allocation to clients requesting one of these services.

동일한 데이터베이스를 용도별로 다른 Service Name으로 노출한다. DBA는 각 서비스에 다른 리소스 제한을 설정할 수 있다. 예를 들어 REPORT 서비스는 CPU를 20%만 사용하도록 제한하여 OLTP 서비스에 영향을 주지 않게 한다.

4.3 연결 수립 과정에서 Service Name의 역할

클라이언트가 연결을 요청할 때의 흐름:



Listener는 Service Name을 기준으로 등록된 Instance와 Service Handler 목록을 조회한다. 여러 Instance가 동일 서비스를 제공하면 로드 밸런싱 알고리즘에 따라 적절한 Instance를 선택한다.

4.4 JDBC Connection String과 Service Name

Spring Boot 환경에서 Service Name이 사용되는 위치:

```
# application.yml
spring:
  datasource:
    # 형식 1: Service Name 사용 (권장)
    url: jdbc:oracle:thin:@//hostname:1521/ORCL

    # 형식 2: SID 사용 (레거시)
    url: jdbc:oracle:thin:@hostname:1521:ORCL
```

두 형식의 차이:

구분	Service Name 방식	SID 방식
구문	`@//host:port/service_name`	`@host:port:SID`
대상	논리적 서비스	물리적 Instance
RAC 지원	가능 (로드 밸런싱, Failover)	불가 (특정 Instance 고정)
권장 여부	권장	레거시 호환용

슬래시(`/`) 뒤에 오는 것이 Service Name이고, 콜론(`:`) 뒤에 오는 것이 SID다. 현대 환경에서는 Service Name 방식을 사용해야 한다.

4.5 Service Name vs SID vs Instance Name

혼동하기 쉬운 세 가지 개념의 차이:

개념	정의	예시	설정 위치
Service Name	클라이언트가 연결 시 사용하는 논리적 이름	`orcl.example.com`	`SERVICE_NAMES` 파라미터
SID	Instance를 식별하는 시스템 식별자	`orcl`	`ORACLE_SID` 환경변수
Instance Name	Instance의 고유 이름 (보통 SID와 동일)	`orcl`	`INSTANCE_NAME` 파라미터

단일 Instance 환경에서는 세 값이 비슷하거나 동일한 경우가 많아 혼동이 생긴다. 그러나 RAC 환경에서는 명확히 구분된다:

```
Service Name: ORCL          ← 클라이언트가 사용 (논리적)
      │
      ├──── Instance 1
      │       SID: orcl1       ← Node 1의 물리적 식별자
      │       Instance Name: orcl1
      │
      └──── Instance 2
            SID: orcl2       ← Node 2의 물리적 식별자
            Instance Name: orcl2
```

4.6 백엔드 개발자 관점의 설계적 의의 (So What)

4.6.1 Connection String 작성 시 Service Name 사용

Wherehouse 프로젝트에서 Oracle 연결 설정:

```
spring:
  datasource:
    url: jdbc:oracle:thin:@//localhost:1521/ORCL
    username: wherehouse
    password: ****
    driver-class-name: oracle.jdbc.OracleDriver
```

이 설정에서 `ORCL`이 Service Name이다. Listener는 이 Service Name을 기준으로 적절한 Service Handler(Server Process)를 할당한다.

4.6.2 환경별 Service Name 분리 전략

실무에서 동일 데이터베이스에 여러 Service Name을 등록하는 패턴:

```
[Production Database]
      |
      ├─── Service: PROD_OLTP      ← 실시간 트랜잭션 (리소스 우선)
      |      Connection Pool: 50
      |      CPU Limit: 70%
      |
      ├─── Service: PROD_BATCH     ← 배치 작업 (리소스 제한)
      |      Connection Pool: 10
      |      CPU Limit: 20%
      |
      └─── Service: PROD_REPORT    ← 리포팅 (읽기 전용)
           Connection Pool: 20
           CPU Limit: 10%
```

애플리케이션 용도에 따라 다른 Service Name으로 연결하면, DBA가 각 서비스별로 리소스를 제어할 수 있다. 리포팅 쿼리가 무거워도 OLTP 서비스에 영향을 주지 않는다.

4.6.3 면접 대응

"Service Name이 뭔가요?"라는 질문에 대한 답변 구조:

1. **정의**: 클라이언트가 데이터베이스 연결 시 사용하는 논리적 식별자
2. **역할**: Listener가 연결 요청을 적절한 Instance/Service Handler로 라우팅하는 기준
3. **SID와의 차이**: SID는 물리적 Instance 식별자, Service Name은 논리적 서비스 식별자
4. **실무적 의미**: RAC 환경 로드 밸런싱, 용도별 리소스 분리, 물리 구조 변경 시 클라이언트 코드 무변경

4.7 정리

Service Name은 물리적 데이터베이스 구조를 추상화하는 논리적 식별자다. 클라이언트는 호스트 주소나 Instance 이름 대신 Service Name으로 연결을 요청하고, Listener가 해당 서비스를 제공하는 Instance 중 적절한 것을 선택하여 연결을 수립한다. 이 추상화 덕분에 물리적 구조가 변경되어도(서버 증설, 장애 복구, RAC 전환 등) 애플리케이션 코드는 수정할 필요가 없다. JDBC Connection String에서 `@//host:port/service_name` 형식으로 사용하며, 레거시 SID 방식(`@host:port:SID`)보다 권장된다.

---

5. Dedicated Server Architecture — 아키텍처적 분석

5.1 정의와 존재 이유 (Why)

공식 문서의 정의:

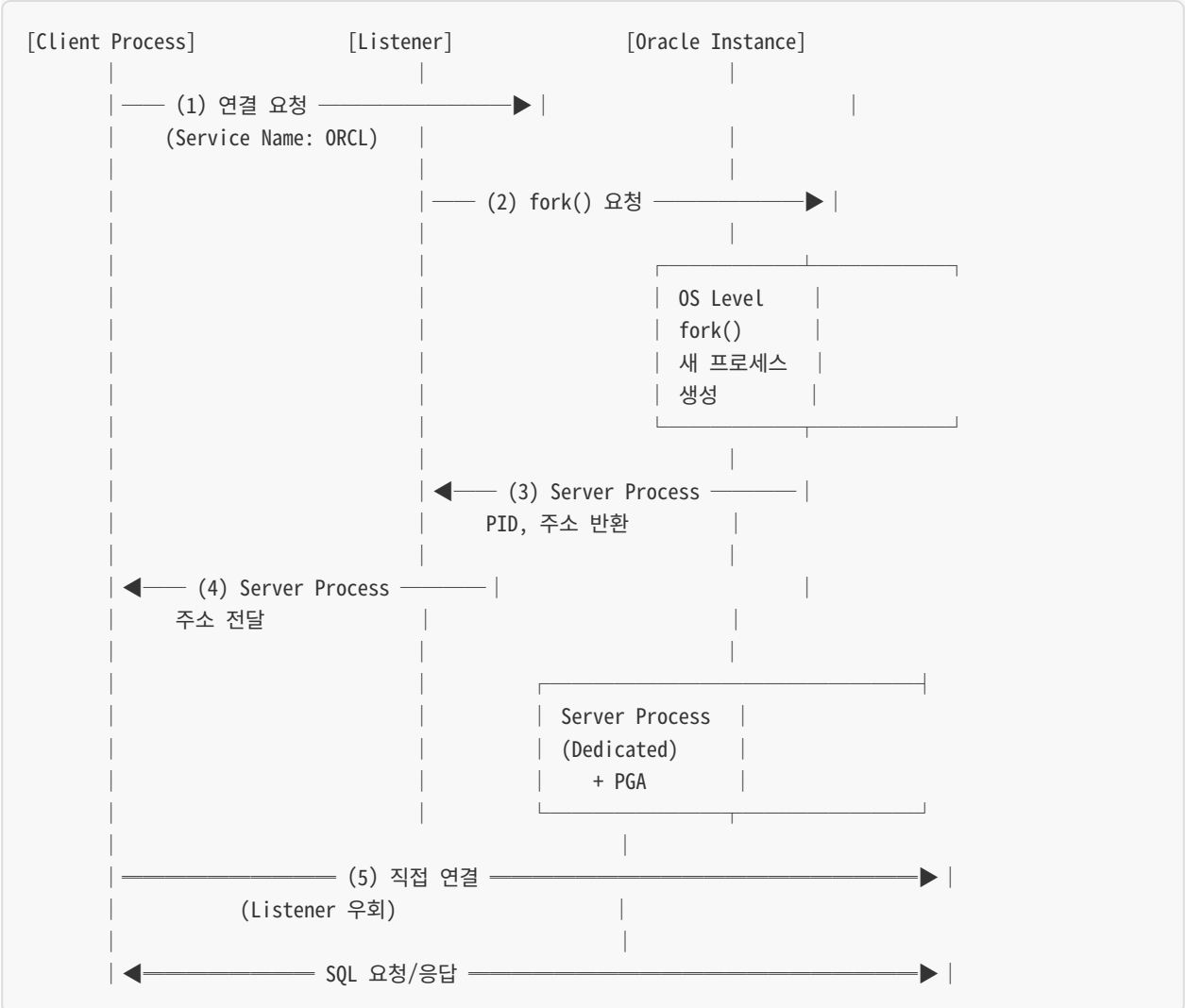
> In a dedicated server architecture, the server process created on behalf of each client process is called a dedicated server process (or shadow process). A dedicated server process is separate from the client process and acts only on its behalf.
> A one-to-one ratio exists between the client processes and server processes. Even when the user is not actively making a database request, the dedicated server process remains—although it is inactive and can be paged out on some operating systems.

Dedicated Server Architecture는 클라이언트 하나당 Server Process 하나를 전용으로 할당하는 구조다. "Dedicated"라는 이름 그대로, 해당 Server Process는 오직 한 클라이언트만을 위해 존재한다.

이 아키텍처가 기본값인 이유는 단순성과 격리성 때문이다. 각 클라이언트의 작업이 독립적인 프로세스에서 수행되므로, 한 클라이언트의 문제(무한 루프, 메모리 누수 등)가 다른 클라이언트에 영향을 주지 않는다. 또한 구현이 단순하여 디버깅과 모니터링이 용이하다.

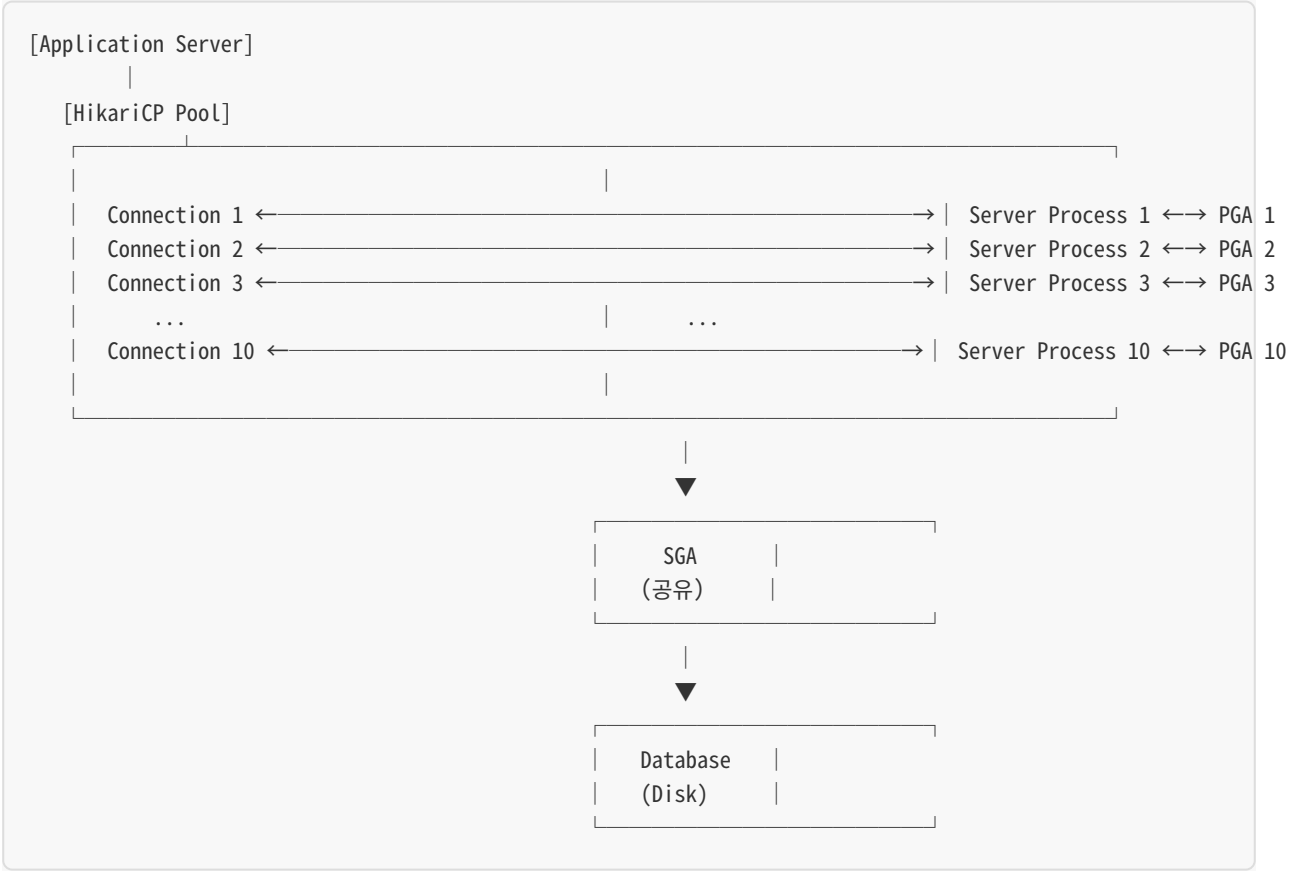
5.2 동작 메커니즘 (How)

5.2.1 연결 수립 과정



핵심은 **(2)번 단계의 fork()**다. Listener가 OS에 새로운 프로세스 생성을 요청하고, 이 프로세스가 해당 클라이언트 전용 Server Process가 된다.

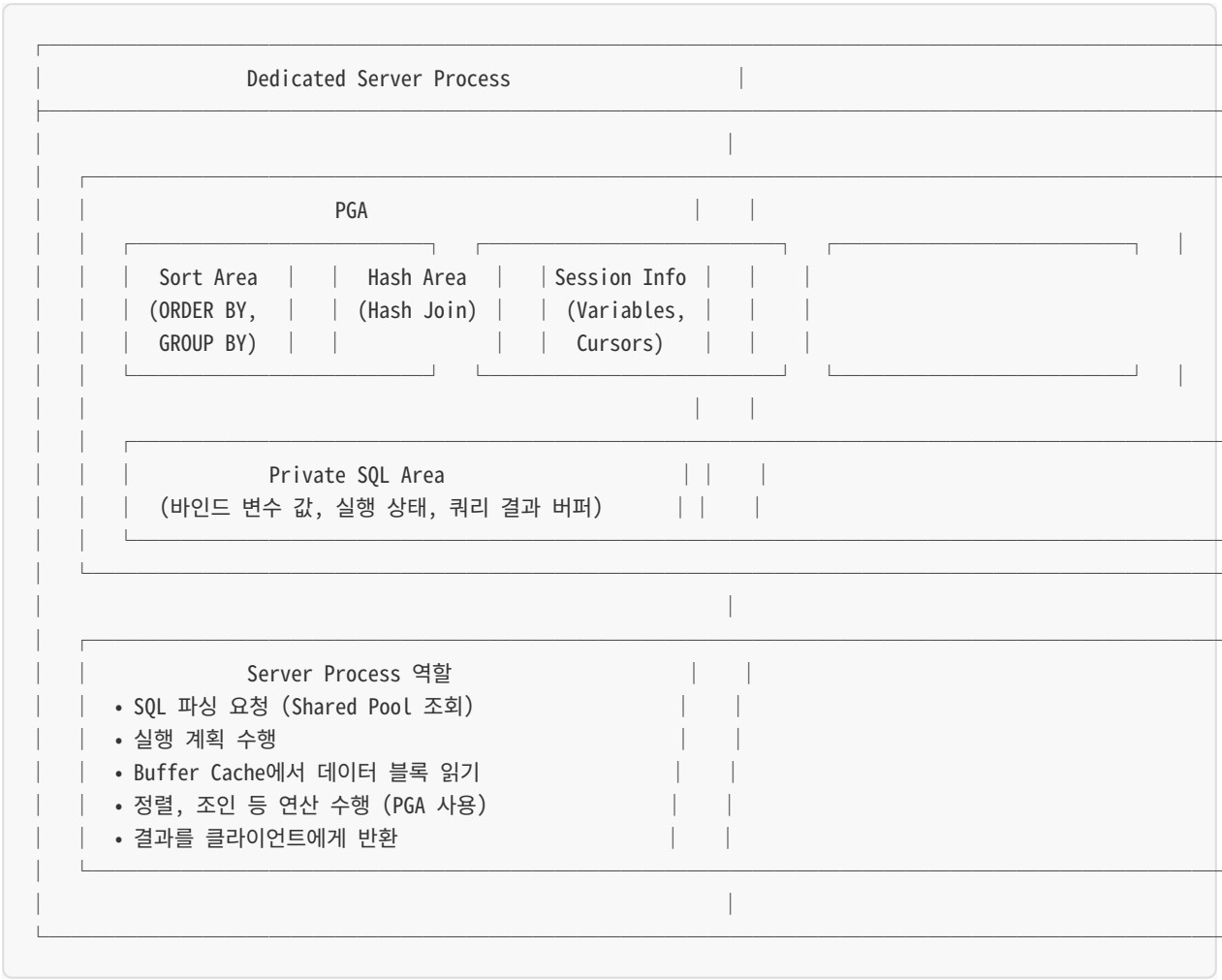
5.2.2 1:1 매핑 구조



Connection Pool에 10개의 Connection이 있으면, Oracle 측에도 정확히 10개의 Dedicated Server Process가 존재한다. 각 Server Process는 자신만의 PGA를 갖고, SGA는 모든 Server Process가 공유한다.

5.2.3 Server Process의 내부 구조

각 Dedicated Server Process가 담당하는 작업:



5.3 Connection 생성 비용의 상세 내역

공식 문서에서 암시하는 비용 구조를 풀어보면:

1단계: TCP 3-Way Handshake

```
Client → SYN → Listener
Client ← SYN+ACK ← Listener
Client → ACK → Listener
```

네트워크 왕복 1.5회, 일반적으로 0.5~2ms

2단계: Oracle Net Services 프로토콜 협상

```
Client → Connect Request (Service Name, Protocol Version)
Client ← Accept Response (Protocol Parameters)
```

Oracle Net 레벨의 핸드셰이크, 1~5ms

3단계: 인증

```
Client → Authentication Request (Username, Password)
Server → Password Verification (SGA의 Data Dictionary 조회)
Client ← Authentication Response
```

암호 검증 및 권한 확인, 5~20ms

4단계: Server Process Fork (가장 비용이 큰 단계)

```
Listener → OS: fork() 시스템 콜
OS: 새 프로세스 생성
    - 프로세스 테이블 엔트리 할당
    - 메모리 공간 복제 (Copy-on-Write)
    - 파일 디스크립터 복제
    - 시그널 핸들러 설정
Server Process: 초기화
    - Oracle 코드 로드
    - SGA 접근 권한 획득
```

OS 레벨 프로세스 생성, 10~50ms

5단계: PGA 할당

```
Server Process: PGA 메모리 영역 할당
    - Sort Area (pga_aggregate_target에 따라 동적)
    - Hash Area
    - Session Memory
    - Private SQL Area
```

메모리 할당, 5~15ms

6단계: 세션 초기화

```
Server Process: 세션 정보 설정
    - NLS 설정 (언어, 날짜 포맷)
    - 세션 변수 초기화
    - Audit 설정
```

세션 컨텍스트 초기화, 2~10ms

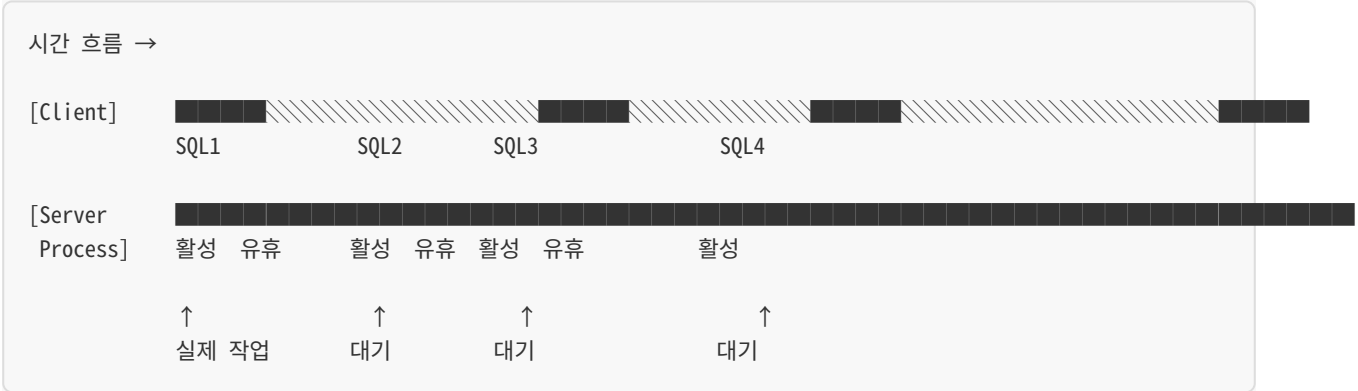
총 비용: 약 50~200ms

5.4 유휴 상태에서의 리소스 점유

공식 문서의 핵심 문장:

> Even when the user is not actively making a database request, the dedicated server process remains—although it is inactive and can be paged out on some operating systems.
> Underutilized dedicated servers sometimes result in inefficient use of operating system resources. Consider an order entry system with dedicated server processes. A customer places an order as a clerk enters the order into the database. For most of the transaction, the clerk is talking to the customer while the server process dedicated to the clerk's client process is idle. The server process is not needed during most of the transaction.

이것이 Dedicated Server의 구조적 비효율성이다. 클라이언트가 SQL을 실행하지 않는 동안에도 Server Process는 유지된다.

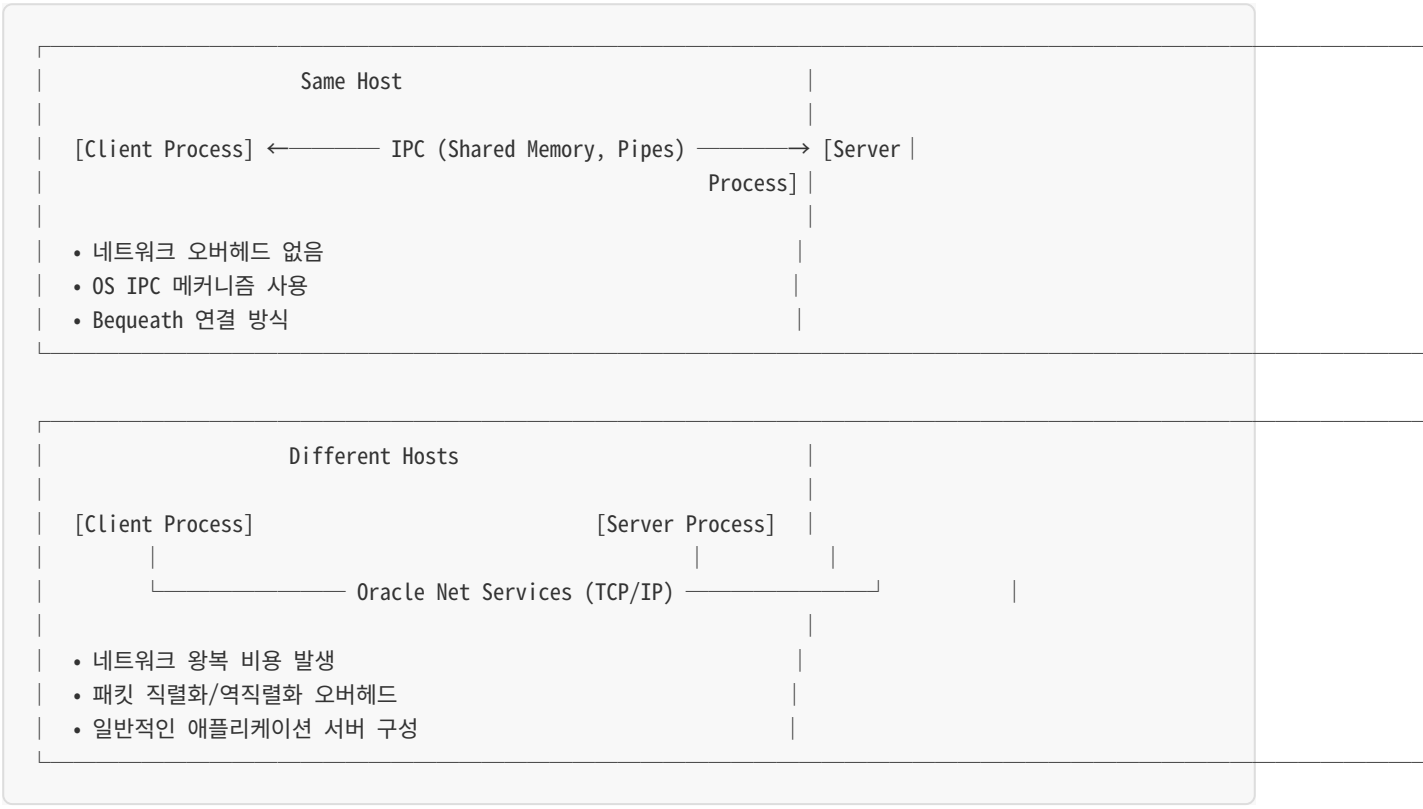


위 다이어그램에서 Server Process가 실제로 일하는 시간(활성)은 전체의 일부에 불과하다. 나머지 시간(유휴) 동안에도 프로세스와 PGA 메모리는 유지된다.

5.5 통신 메커니즘

공식 문서의 설명:

> In the dedicated server architecture, the user and server processes communicate using different mechanisms:
> - If the client process and the dedicated server process run on the same computer, then the program interface uses the host operating system's interprocess communication mechanism to perform its job.
> - If the client process and the dedicated server process run on different computers, then the program interface provides the communication mechanisms (such as the network software and Oracle Net Services) between the programs.



Wherehouse 프로젝트처럼 애플리케이션 서버와 DB 서버가 분리된 환경에서는 모든 SQL 호출이 네트워크를 경유한다. N+1 문제로 25번 쿼리가 발생하면 25번의 네트워크 왕복이 발생하는 것이다.

5.6 백엔드 개발자 관점의 설계적 의의 (So What)

5.6.1 Connection Pool의 필요성에 대한 아키텍처적 근거

Dedicated Server Architecture에서 Connection Pool 없이 매 요청마다 연결을 생성하면:

```
[HTTP 요청 1] → Connection 생성 (50~200ms) → SQL 실행 (10ms) → Connection 종료
[HTTP 요청 2] → Connection 생성 (50~200ms) → SQL 실행 (10ms) → Connection 종료
[HTTP 요청 3] → Connection 생성 (50~200ms) → SQL 실행 (10ms) → Connection 종료
```

실제 SQL 실행 시간보다 Connection 생성 비용이 5~20배 더 크다. Connection Pool을 사용하면:

```
[애플리케이션 시작] → Connection 10개 생성 (1회성 비용)

[HTTP 요청 1] → Pool에서 Connection 획득 (0.1ms) → SQL 실행 (10ms) → 반환
[HTTP 요청 2] → Pool에서 Connection 획득 (0.1ms) → SQL 실행 (10ms) → 반환
[HTTP 요청 3] → Pool에서 Connection 획득 (0.1ms) → SQL 실행 (10ms) → 반환
```

Connection 획득 비용이 0.1ms 수준으로 감소한다.

5.6.2 Wherehouse 프로젝트 테스트 데이터 재해석

1차 테스트 결과를 Dedicated Server Architecture 관점에서 분석:

지표	값	아키텍처적 의미
HikariCP Pool Size	10	Dedicated Server Process 10개 유지
RDB 조회 시간	343ms	Server Process의 실제 작업 시간
RDB 시간 비율	6.2%	Server Process 활용률
총 소요 시간	5,531ms	나머지 93.8%는 어디서 소요?
HikariCP Waiting	최대 9건	Connection(Server Process) 획득 경합

RDB 시간 비율 6.2%의 의미: Dedicated Server Process가 실제로 SQL을 처리한 시간은 전체의 6.2%에 불과하다. 나머지 93.8%는 애플리케이션 레벨에서 Connection 획득 대기, 네트워크 왕복, 애플리케이션 로직 처리에 소요되었다.

N+1로 25번 쿼리가 발생할 때 (@Transactional 범위 내):

```
[Request Thread]
      |
      ├──── @Transactional 시작
      |          |
      |          ├──── Connection 획득 (1회) ← Pool에서 대기 가능 시점
      |          |          ↓
      |          |    [동일 Server Process에 25번 RTT 발생]
      |          |          |
      |          |          ├──── SQL 1: SELECT ... WHERE property_id = 1
      |          |          |           ↓ (RTT: ~13.7ms)
      |          |          |          결과 반환
      |          |          |
      |          |          ├──── SQL 2: SELECT ... WHERE property_id = 2
      |          |          |           ↓ (RTT: ~13.7ms)
      |          |          |          결과 반환
      |          |          |
      |          |          |    ... (23번 더 반복)
      |          |          |
      |          |          └──── SQL 25: SELECT ... WHERE property_id = 25
      |          |                      ↓ (RTT: ~13.7ms)
      |          |                     결과 반환
      |          |
      |          |    총 RTT: 13.7ms × 25 = 342.5ms (≈ RDB 조회 시간 343ms)
      |          |
      |          └──── Connection 반환 (1회)
      |
      └──── @Transactional 종료
```

@Transactional 범위 내에서 Connection은 ThreadLocal에 바인딩되어 트랜잭션 종료까지 유지된다. 따라서 25번 발생하는 것은 Connection 획득/반환 사이클이 아니라 **네트워크 왕복(RTT)**이다. 25번의 RTT 누적으로 인해 하나의 요청이 Connection을 오래 점유하게 되고, 이로 인해 동시에 들어오는 다른 HTTP 요청들이 Pool에서 대기하게 된다. HikariCP Waiting 최대 9건은 이러한 동시 요청 간 경합 상황을 나타낸다.

5.6.3 Pool 크기 결정의 아키텍처적 고려사항

Dedicated Server Architecture에서 Pool 크기는 곧 Server Process 수다:

```
Pool 크기 증가 시:
  장점: Connection 경합 감소, 동시 처리량 증가
  단점: PGA 메모리 총량 증가, OS 프로세스 관리 오버헤드 증가

Pool 크기 감소 시:
  장점: 서버 리소스 절약
  단점: Connection 경합 증가, 대기 시간 증가
```

Wherehouse 프로젝트의 경우, Pool 크기를 늘리는 것보다 N+1 문제를 해결하여 쿼리 횟수를 줄이는 것이 근본적인 해결책이었다. 3차 테스트에서 Chunk 방식으로 쿼리 횟수를 9회로 줄이자, Connection 경합 자체가 감소했다.

5.6.4 면접 대응

"Dedicated Server Architecture가 뭔가요?"에 대한 답변 구조:

1. **정의**: 클라이언트 Connection 하나당 Server Process 하나를 전용으로 할당하는 Oracle의 기본 연결 아키텍처
2. **특징**: 1:1 매핑, 클라이언트 유휴 시에도 Server Process 유지, 각 Server Process가 독립적인 PGA 보유
3. **Connection 생성 비용**: Server Process 생성, PGA 할당, 세션 초기화 등으로 일반적으로 수십~수백 ms 소요
4. **Connection Pool 필요성**: 생성 비용이 크므로 미리 생성해두고 재사용
5. **프로젝트 연결**: "제 프로젝트에서 N+1로 25번 쿼리가 발생했을 때, Connection Pool 크기 10개에서 경합이 발생했고, 전체 응답 시간의 94%가 Connection 획득 대기, 네트워크 왕복, 역직렬화 등 복합적 요소로 소요되었습니다"

5.7 정리

Dedicated Server Architecture는 클라이언트당 전용 Server Process를 할당하는 1:1 매핑 구조다. 단순하고 격리성이 좋지만, Connection 생성 비용이 크고(Server Process 생성 + PGA 할당) 유휴 상태에서도 리소스를 점유한다. 이 구조적 특성이 Connection Pool의 필요성을 만들어내며, Pool 크기 = Dedicated Server Process 수 = 동시 SQL 처리 능력이라는 등식이 성립한다. Wherehouse 프로젝트에서 RDB 시간 비율이 6.2%밖에 안 되는데 전체 응답이 5초가 넘은 이유는, N+1로 인한 25번의 RTT 누적으로 Connection 점유 시간이 길어지고, 동시 요청 간 Pool 경합이 발생했으며, 네트워크 왕복과 역직렬화 등이 복합적으로 작용했기 때문이다.

---

6. Shared Server Architecture 학습 필요성 판단

6.1 결론: 깊은 학습 불필요, 비교 관점의 핵심만 파악

6.2 학습 불필요 판단 근거

6.2.1 현대 웹 애플리케이션 아키텍처와 맞지 않음

Spring Boot + HikariCP 환경에서는 애플리케이션 레벨에서 Connection Pooling을 수행한다. Shared Server는 Oracle 내부에서 Server Process를 공유하는 방식인데, 이미 HikariCP가 Connection 재사용을 담당하므로 Shared Server의 이점이 상쇄된다.

```
[현대 웹 애플리케이션 구조]

[Tomcat Thread Pool] → [HikariCP Pool] → [Dedicated Server Process]
       ↑                    ↑                    ↑
   요청 레벨 공유        Connection 재사용      1:1 매핑 유지

→ 애플리케이션 레벨에서 이미 풀링하므로 Shared Server 불필요
```

6.2.2 Shared Server의 주 사용 사례가 목적과 다름

Shared Server는 수천~수만 개의 동시 연결이 필요한 특수 환경(대규모 OLTP, 레거시 2-tier 아키텍처)에서 사용한다. Wherehouse 프로젝트나 일반적인 Spring Boot 애플리케이션은 이 범주에 해당하지 않는다.

6.2.3 설정과 튜닝이 DBA 영역

Shared Server 활성화, Dispatcher 수 설정, Request Queue 튜닝 등은 DBA가 담당한다. 백엔드 개발자가 직접 다룰 일이 거의 없다.

6.3 비교 관점에서 알아두면 좋은 핵심 차이

면접에서 "Dedicated Server와 Shared Server의 차이"를 물어볼 수 있으므로, 핵심만 정리:

구분	Dedicated Server	Shared Server
Server Process 할당	클라이언트당 1개 (1:1)	여러 클라이언트가 공유 (N:M)
연결 단위	Connection 단위	Request 단위
UGA 위치	PGA (프로세스별 독립)	SGA (공유 메모리)
유휴 시 리소스	Server Process 유지	Server Process 반환
적합한 환경	일반적인 웹 애플리케이션	대규모 동시 연결, 짧은 트랜잭션
현대 사용 빈도	대부분 (기본값)	드묾

6.4 대신 학습 권장: Database Resident Connection Pooling (DRCP)

Shared Server보다 DRCP가 학습 가치가 높다.

권장 이유:

- **HikariCP와 비교 가능**: 애플리케이션 측 Pool(HikariCP)과 DB 측 Pool(DRCP)의 차이를 이해할 수 있다
- **현대적인 접근**: Oracle 11g 이후 도입된 메커니즘으로, Shared Server의 단점을 보완한 구조다
- **면접 차별화**: "Oracle의 DRCP와 HikariCP의 역할 차이를 아는가?"라는 질문에 답할 수 있다

공식 문서의 DRCP 설명:

> Database Resident Connection Pooling (DRCP) provides a connection pool of dedicated servers for typical Web application scenarios... DRCP uses a pooled server, which is the equivalent of a dedicated server process and a database session combined.

DRCP는 Dedicated Server의 장점(독립적 PGA, 단순한 구조)을 유지하면서 Oracle 측에서도 Server Process를 풀링한다. PHP처럼 요청마다 새 프로세스를 생성하는 환경에서 유용하다.

---

7. Database Resident Connection Pooling (DRCP) — 아키텍처적 분석

7.1 정의와 존재 이유 (Why)

공식 문서의 정의:

> Database Resident Connection Pooling (DRCP) provides a connection pool of dedicated servers for typical Web application scenarios.
> DRCP uses a pooled server, which is the equivalent of a dedicated server process (not a shared server process) and a database session combined. The pooled server model avoids the overhead of dedicating a server for every connection that requires the server for a short period.

DRCP는 Oracle Database 내부에서 Server Process를 풀링하는 메커니즘이다. HikariCP가 애플리케이션 측에서 Connection을 풀링한다면, DRCP는 데이터베이스 측에서 Server Process를 풀링한다.

DRCP가 필요한 이유는 Dedicated Server의 구조적 한계 때문이다. Dedicated Server 모드에서는 Connection 하나당 Server Process 하나가 할당된다. 1,000개의 Connection이 필요하면 1,000개의 Server Process가 생성되어야 하고, 이는 막대한 메모리(PGA)와 OS 리소스를 소모한다. 대부분의 Connection이 유휴 상태임에도 Server Process는 유지된다.

```
[Dedicated Server의 한계]

Connection 1,000개 × PGA 5MB = 5GB 메모리
                   + OS 프로세스 관리 오버헤드
                   + Context Switching 비용

→ 실제 동시 활성 쿼리는 50개 수준인데 1,000개 프로세스 유지
```

DRCP는 이 문제를 해결한다. 1,000개의 Connection이 있어도 실제 Server Process는 50~100개만 유지하고, 필요할 때만 할당한다.

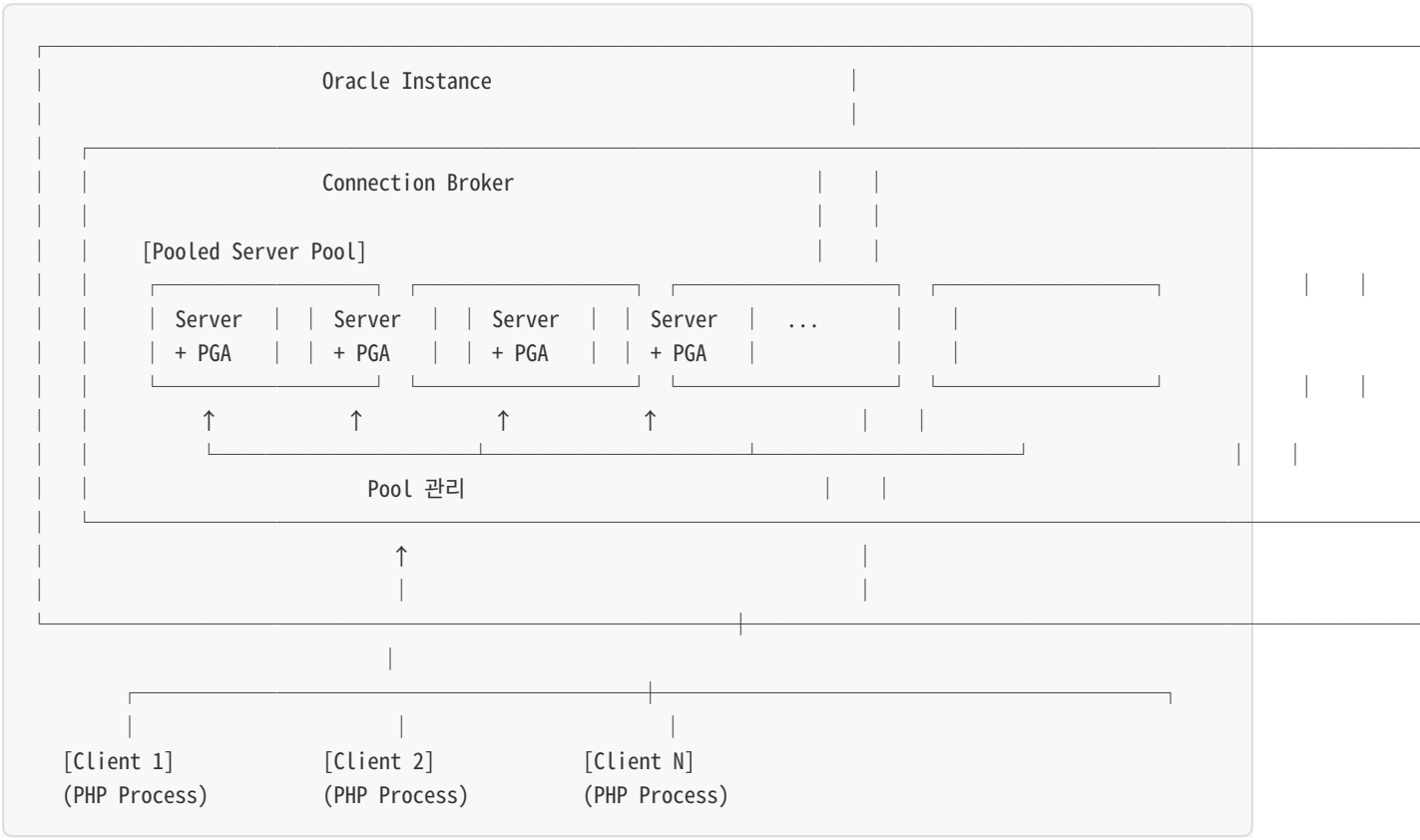
7.2 동작 메커니즘 (How)

공식 문서의 핵심 설명:

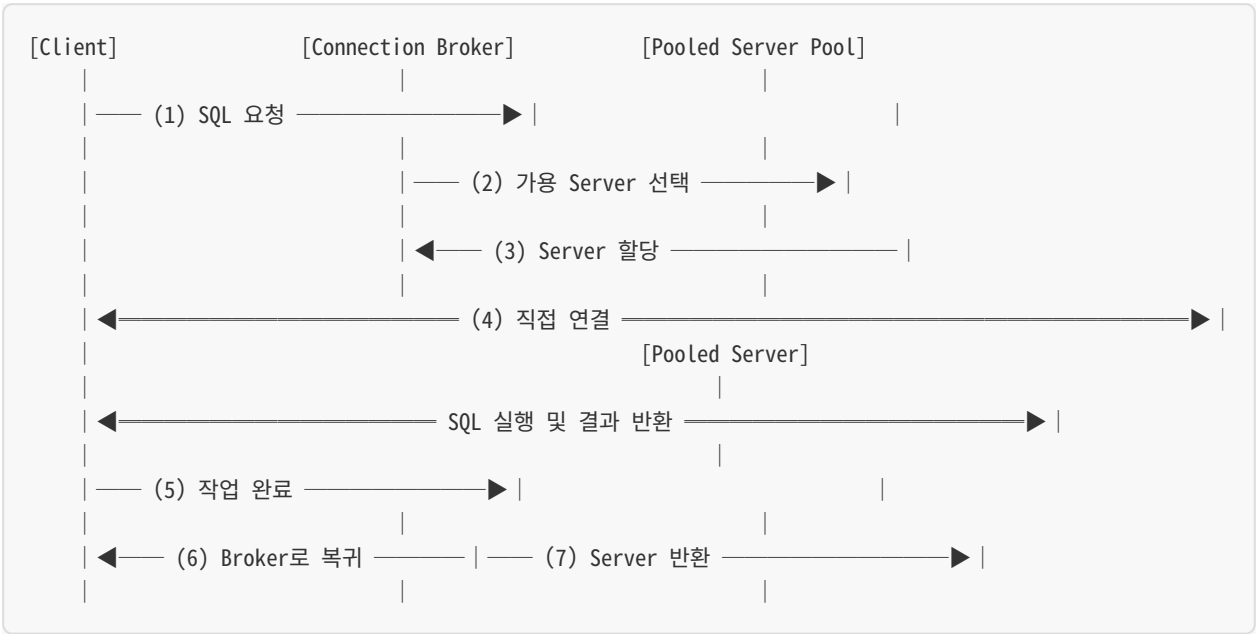
> Clients obtaining connections from the database resident connection pool connect to an Oracle background process known as the connection broker. The connection broker implements the pool functionality and multiplexes pooled servers among inbound connections from client processes.
> When a client requires database access, the connection broker picks up a server process from the pool and hands it off to the client. The client is directly connected to the server process until the request is served. After the server has finished, the server process is released into the pool. The connection from the client is restored to the broker.

7.2.1 핵심 컴포넌트: Connection Broker

Connection Broker는 DRCP의 핵심 프로세스다. 클라이언트 연결을 받아 Pooled Server를 할당하고 회수하는 역할을 한다.



7.2.2 요청 처리 흐름



핵심은 (4)번 단계다. 클라이언트가 Pooled Server에 직접 연결된다. Shared Server처럼 Request Queue를 거치지 않으므로 큐잉 오버헤드가 없다. 작업이 완료되면 Server Process는 Pool로 반환되고, 클라이언트 연결은 Broker로 복귀한다.

7.2.3 Pooled Server의 특성

공식 문서:

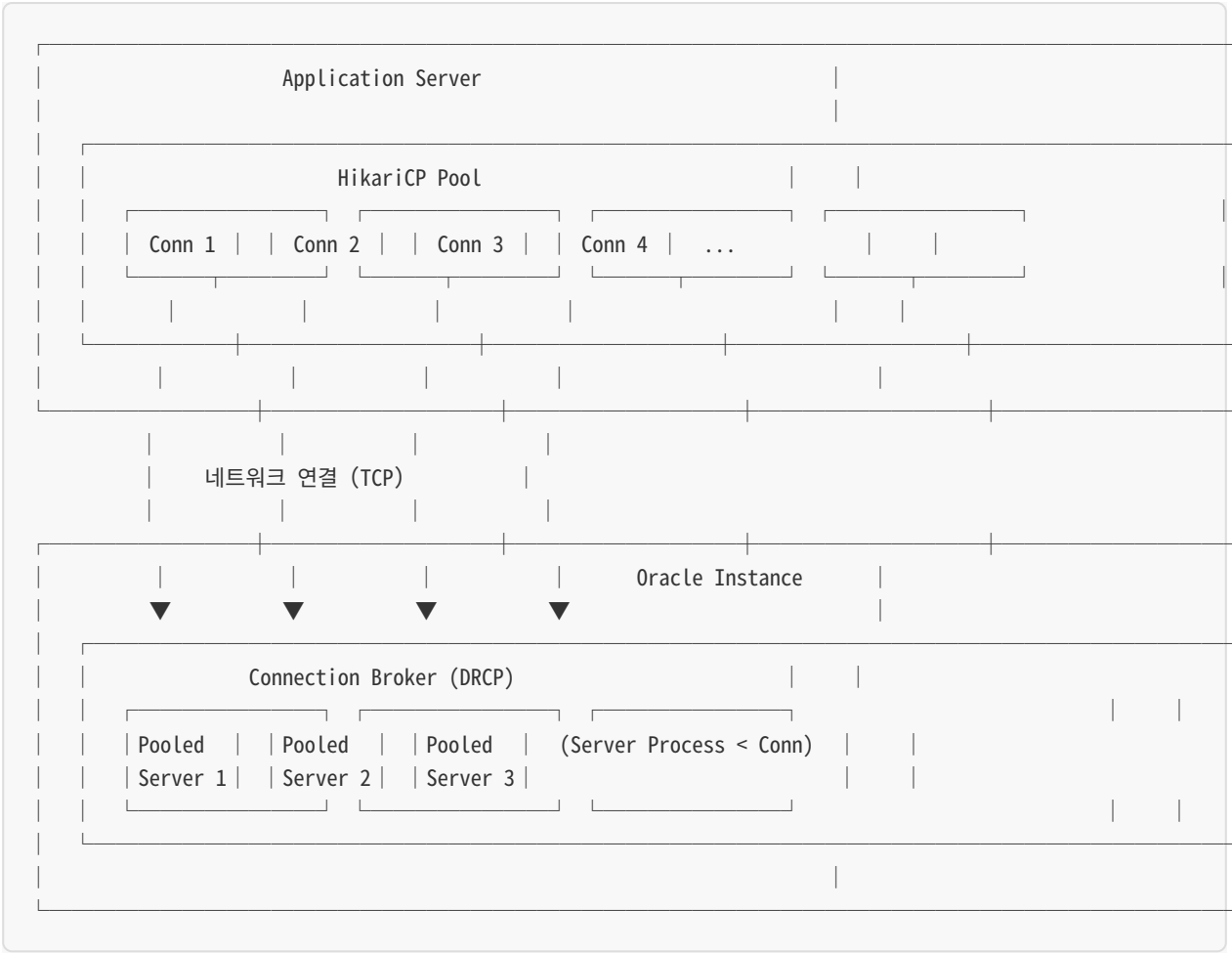
> In DRCP, releasing resources leaves the session intact, but no longer associated with a connection (server process). Unlike in shared server, this session stores its UGA in the PGA, not in the SGA. A client can reestablish a connection transparently upon detecting activity.

특성	DRCP (Pooled Server)	Shared Server
Server Process 유형	Dedicated Server와 동일	Shared Server Process
UGA 위치	PGA (프로세스별)	SGA (공유 메모리)
요청 처리 방식	직접 연결	Request Queue 경유
세션 상태	클라이언트별 독립 유지	제한적

DRCP는 Dedicated Server의 장점(독립 PGA, 직접 연결)을 유지하면서 Server Process만 공유한다.

7.3 DRCP vs HikariCP: 풀링 레이어의 차이

두 풀링 메커니즘은 다른 레이어에서 작동한다.



- **HikariCP의 풀링 대상**: TCP Connection (네트워크 연결 + Oracle Session)
- **DRCP의 풀링 대상**: Server Process (OS 프로세스 + PGA 메모리)

구분	HikariCP	DRCP
풀링 위치	애플리케이션 서버	데이터베이스 서버
풀링 대상	Connection (네트워크 연결)	Server Process (OS 프로세스)
절약 대상	Connection 생성 비용	Server Process 생성 비용, 메모리
관리 주체	개발자	DBA
설정 방식	application.yml	Oracle 파라미터

7.4 DRCP가 효과적인 환경

공식 문서:

> DRCP provides a connection pool of dedicated servers for typical Web application scenarios... Complements middle-tier connection pools that share connections between threads in a middle-tier process... Provides pooling for architectures with multi-process, single-threaded application servers, such as PHP and Apache, that cannot do middle-tier connection pooling.

DRCP가 효과적인 경우:

- **PHP + Apache 환경**: 요청마다 새 프로세스 생성, 애플리케이션 측 풀링 불가
- **대규모 연결이 필요하지만 동시 활성 쿼리는 적은 환경**
- **여러 애플리케이션 서버가 동일 DB에 연결하는 환경**

```
[PHP 환경에서 DRCP의 효과]

Without DRCP:
  Apache Worker 1,000개 × Dedicated Server 1,000개 × PGA 5MB = 5GB

With DRCP:
  Apache Worker 1,000개 → Connection Broker → Pooled Server 100개 × PGA 5MB = 500MB

  → 메모리 90% 절약
```

DRCP가 불필요한 경우:

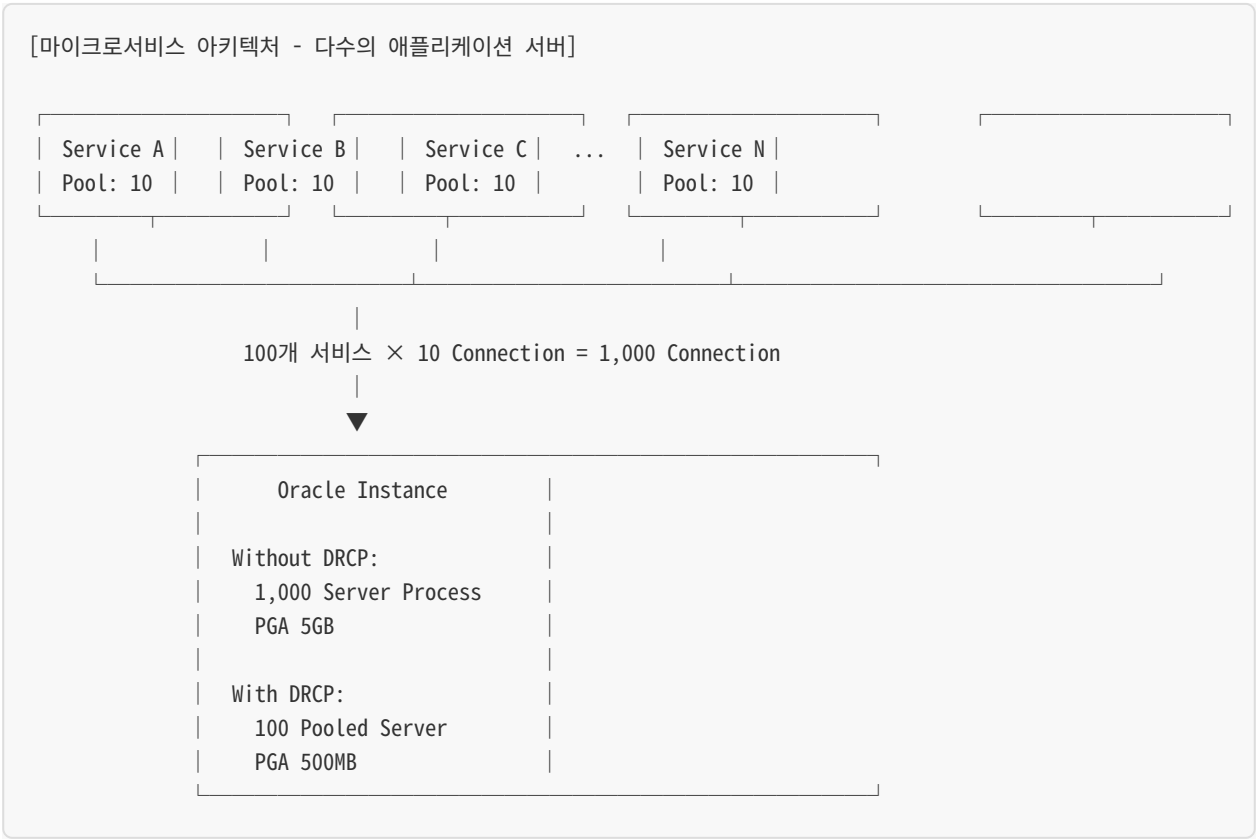
```
[Spring Boot + HikariCP 환경]

HikariCP Pool 10개 → Dedicated Server 10개 → PGA 50MB

→ 이미 적은 수의 Connection만 유지
→ DRCP 추가 효과 미미
→ 오히려 Connection Broker 오버헤드 발생 가능
```

7.5 Spring Boot 환경에서의 DRCP 적용 여부

일반적인 Spring Boot 애플리케이션: DRCP 불필요

이유:
- **HikariCP가 이미 Connection 풀링 수행**: Connection 수가 이미 제한됨 (보통 10~50개)
- **Dedicated Server 수가 적음**: PGA 메모리 부담이 크지 않음
- **추가 복잡성**: DRCP 활성화, Connection Class 설정 등 관리 포인트 증가

DRCP가 의미 있는 Spring Boot 환경:



7.6 백엔드 개발자 관점의 설계적 의의 (So What)

7.6.1 Wherehouse 프로젝트에서 DRCP 필요성

현재 구조:
- HikariCP Pool 크기: 10
- Oracle 측 Dedicated Server: 10개
- PGA 총량: 약 50MB (가정)

이 규모에서 DRCP는 불필요하다. Server Process 10개 정도는 Oracle 서버가 충분히 감당할 수 있다.

DRCP가 필요해지는 시점:

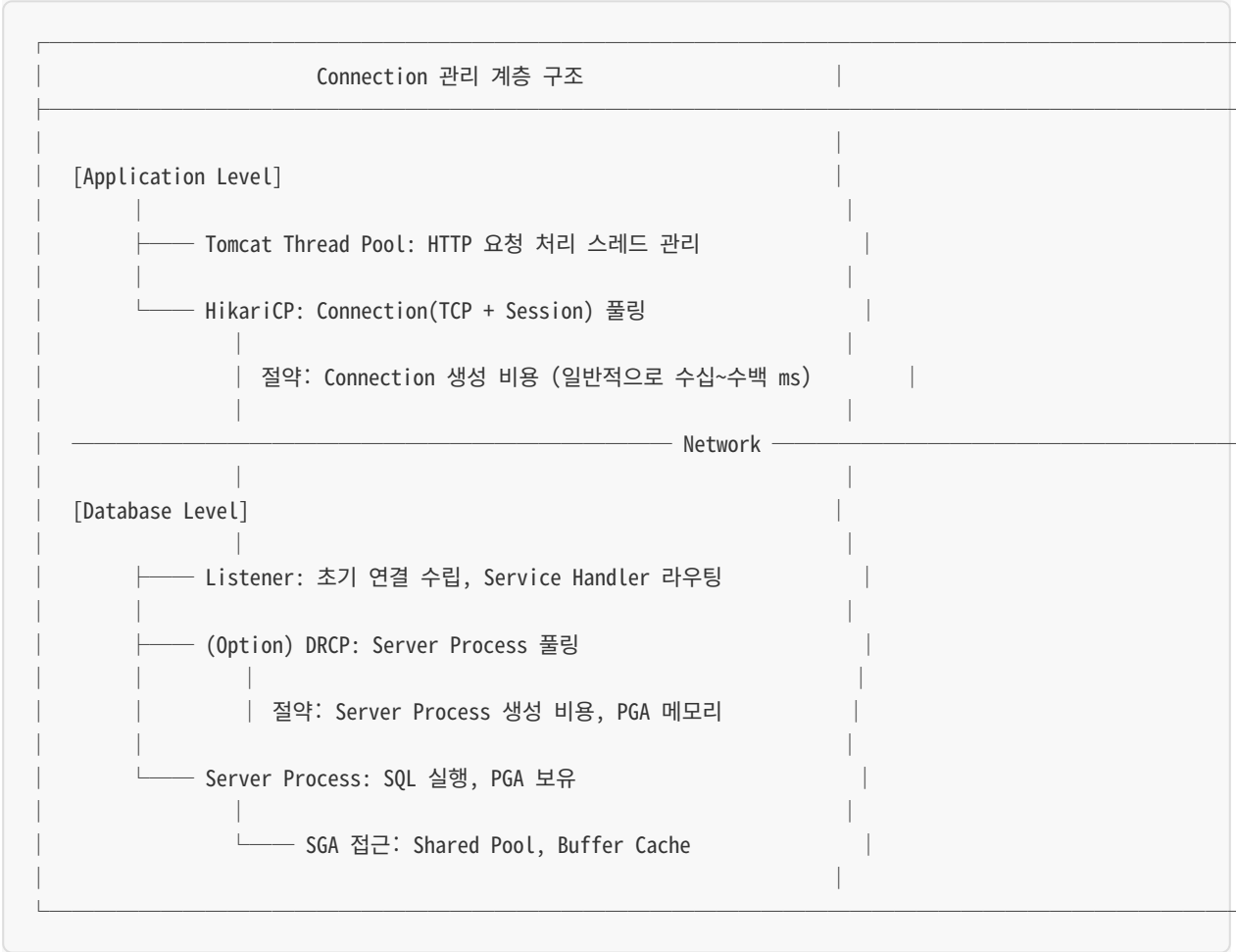
```
단일 애플리케이션 → DRCP 불필요
       ↓
서비스 분리 (5개 마이크로서비스) → 검토 필요
       ↓
대규모 마이크로서비스 (50개 이상) → DRCP 권장
```

7.6.2 면접에서의 활용

"Oracle의 DRCP를 알고 계신가요?"라는 질문에 대한 답변 구조:

1. **정의**: Oracle 내부에서 Server Process를 풀링하는 메커니즘
2. **HikariCP와의 차이**: HikariCP는 애플리케이션 측에서 Connection을 풀링하고, DRCP는 데이터베이스 측에서 Server Process를 풀링한다
3. **적합한 환경**: PHP처럼 애플리케이션 측 풀링이 불가능한 환경, 또는 다수의 애플리케이션 서버가 동일 DB에 연결하는 마이크로서비스 환경
4. **불필요한 환경**: 단일 Spring Boot 애플리케이션처럼 이미 HikariCP로 적은 수의 Connection만 유지하는 경우

7.7 Connection Pool 아키텍처 전체 그림



7.8 정리

DRCP는 Oracle 내부에서 Server Process를 풀링하는 메커니즘으로, Dedicated Server의 장점(독립 PGA, 직접 연결)을 유지하면서 Server Process를 공유한다. HikariCP가 애플리케이션 측에서 Connection을 풀링한다면, DRCP는 데이터베이스 측에서 Server Process를 풀링한다. PHP처럼 애플리케이션 측 풀링이 불가능한 환경이나, 다수의 마이크로서비스가 동일 DB에 연결하는 환경에서 효과적이다. 단일 Spring Boot 애플리케이션에서는 HikariCP만으로 충분하며 DRCP의 추가 효과가 미미하다.

---

8. DRCP 심화 학습 필요성 판단

8.1 결론: 불필요. 현재 개요 수준으로 충분하다.

8.2 판단 근거

8.2.1 실무 적용 가능성이 낮다

Wherehouse 프로젝트 환경(Spring Boot + HikariCP + Oracle)에서 DRCP를 적용할 이유가 없다. HikariCP Pool 크기 10개 수준에서 Dedicated Server 10개는 Oracle 서버가 충분히 감당한다. DRCP를 활성화하면 오히려 Connection Broker 경유 오버헤드만 추가된다.

DRCP 심화 내용(Connection Class, Purity Level, 파라미터 튜닝)을 알아도 적용할 프로젝트가 없다면 면접에서 어필하기 어렵다.

8.2.2 면접 질문 빈도가 낮다

백엔드 개발자 면접에서 DRCP 상세를 물어볼 가능성은 극히 낮다. 예상되는 질문 수준:

질문 빈도	예시
높음	"Connection Pool이 왜 필요한가요?"
중간	"HikariCP 설정 중 중요한 것은?"
낮음	"Dedicated Server와 Shared Server 차이는?"
극히 낮음	"DRCP의 Connection Class가 뭔가요?"

DRCP 심화는 DBA 면접 영역이다.

8.2.3 학습 ROI가 낮다

DRCP 심화 학습 시 다뤄야 할 내용:
- Connection Class 개념과 설정
- Purity Level (SELF, NEW)

- DRCP Pool 파라미터 튜닝 (MINSIZE, MAXSIZE, INCRSIZE)
- DRCP 모니터링 뷰 (V$CPOOL_STATS)

이 내용을 학습하는 데 2~3시간이 소요되지만, 면접에서 활용할 확률은 5% 미만이다.

8.3 대신 시간을 투자해야 할 영역

8.3.1 이미 학습한 내용과 프로젝트 데이터 연결 연습

"데이터 해석 연습 가이드"가 핵심이다. 아키텍처 개념을 아는 것과 그것을 본인 데이터로 설명하는 것은 다르다.

```
[연습 예시]

Q: "1차 테스트에서 RDB 시간 비율이 6.2%인데 나머지 94%는 어디서 소요되었는가?"

A: "94%의 구성 요소를 분리하면 다음과 같습니다.

    첫째, Connection 획득 대기입니다. HikariCP Waiting이 최대 9건 발생했고,
    N+1로 25번 쿼리의 RTT가 누적되어 Connection 점유 시간이 길어지면서 동시 요청 간 경합이 발생했습니다.

    둘째, 네트워크 왕복 오버헤드입니다. 25번의 SELECT는 최소 25번의 Oracle Net
    패키징/언패킹 + TCP 왕복을 발생시킵니다.

    셋째, 결과 역직렬화입니다. JDBC ResultSet → Java 객체 변환 비용이 있습니다.

    넷째, 비즈니스 로직입니다. Spring 프레임워크 오버헤드, AOP 프록시 등이 포함됩니다.

    참고로 3차 테스트(Bulk Fetch)에서 Waiting이 0임에도 RDB 외 시간이 71%였습니다.
    이는 Connection 획득 대기만이 아니라 위 요소들이 복합적으로 작용함을 증명합니다."
```

이 연결을 자연스럽게 할 수 있으면 면접 합격선을 넘는다.

8.3.2 HikariCP 설정의 아키텍처적 의미

DRCP보다 HikariCP 파라미터를 깊이 이해하는 것이 실용적이다:

파라미터	아키텍처적 의미
maximumPoolSize	Oracle 측 Dedicated Server 최대 개수
minimumIdle	유휴 시에도 유지할 Server Process 수
connectionTimeout	Connection 획득 대기 최대 시간
maxLifetime	Server Process 재생성 주기

이 파라미터들이 Oracle 아키텍처와 어떻게 연결되는지 설명할 수 있으면 차별화된다.

8.3.3 면접 예상 질문 답변 정리

Oracle 아키텍처 학습의 최종 목표는 면접 대응이다. 다음 질문들에 대한 답변을 정리하는 것이 DRCP 심화보다 ROI가 높다:

- "N+1 문제를 어떻게 발견하셨나요?"
- "왜 Bulk Fetch로 바꿨는데 오히려 느려졌나요?"
- "Connection Pool이 왜 필요한가요?"
- "Oracle에서 쿼리가 실행되는 과정을 설명해주세요"

8.4 최종 권장 학습 경로

```
[완료]
├─── Instance vs Database
├─── SGA 구성요소
├─── Server Process와 PGA
├─── SQL 처리 흐름
├─── Oracle Net Services
│    ├─── Service Handler
│    ├─── Service Names
│    └─── Dedicated Server Architecture
└─── DRCP (개요 수준) ← 현재 위치

[다음 단계 - 권장]
├─── 학습 내용 + 프로젝트 데이터 연결 연습
├─── HikariCP 파라미터와 Oracle 아키텍처 연결
└─── 면접 예상 질문 답변 정리

[스킵]
└─── DRCP 심화 (Connection Class, Purity Level, 튜닝)
```