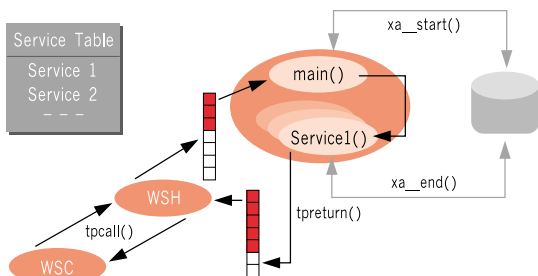
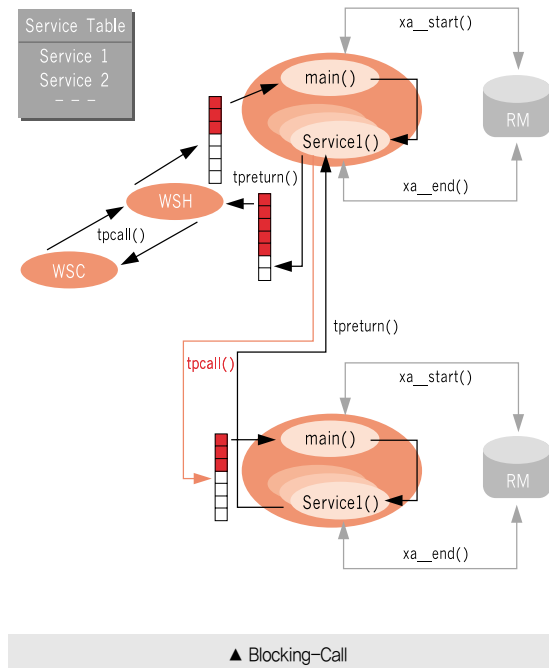


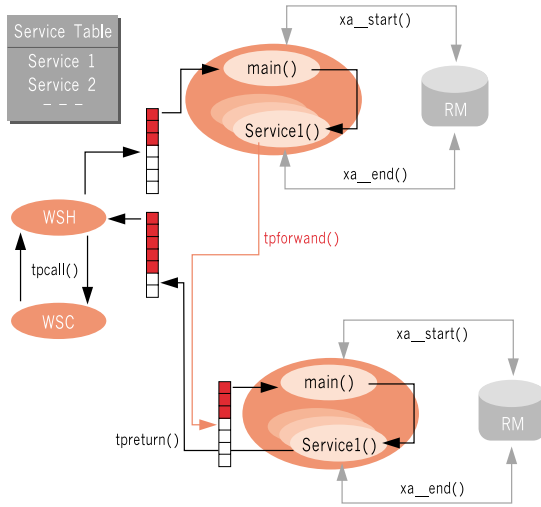
TUXEDO에서의 서비스간의 호출

일반적으로 Client에서 Service를 호출할 때의 상황이 아래 그림에 나타나 있다. Workstation Client(WSC)에서 Service를 호출하면 Workstation Handler(WSH)가 메시지를 Server Process의 Message Queue에 넣는다. WSH 대신에 Native Client나 다른 Server Process가 Service를 호출할 때에도 마찬가지이다. Service Table을 명시한 이유는 WSH에서 Client가 호출한 Service로 요청을 보내기 위해서는 어느 Server가 이 Service를 제공하는지, 이 Server의 Message Queue의 ID는 어떻게 되는지 등등에 대한 정보, 즉 Service에 대한 여러 가지 정보를 알아야 하는데 이 정보를 가지고 있는 곳이 Service Table(Bulletin Board(BB)안에 있다.)이기 때문이다. Server Process의 main() 함수는 Message Queue에서 메시지를 꺼내어 필요로 하는 Service를 제공하는 함수(일반적으로 Service 이름과 이 Service를 수행하는 함수 이름을 같이 사용하지만 다르게 사용할 수도 있다.)로 분기시켜 준다. 분기하기 직전에 만약 Global Transaction에 이 Service가 포함되어 있으면 xa_start()를 호출하여 RM에게 XA Session의 시작을 알린다. Service는 작업을 끝낸 후에 tpreturn() 호출하고 tpreturn()내에서는 필요하면 xa_end()를 호출하여 RM에게 XA Session의 종료로 알린다. Server Process는 반환된 메시지를 WSH가 가지고 있는 Message Queue에 이 메시지를 넣는다. WSH는 이 메시지를 Client에게 반환한다. 이와 같은 과정은 Native Client가 Service를 호출한 경우나 Service에서 다른 Service를 호출할 때에도 마찬가지이다. Message Queue를 사용하는 이유는 Server Process가 특정 Client에 종속되지 않게 하고 동일한 자원을 사용하는 호출이 순차적으로 자원에 접근하도록 하여 충돌을 최소화할 수 있기 때문이다. 그러나 현재 수행중인 Service가 시간을 많이 소모하거나 너무 많은 메시지가 Queue에 저장되어 있으면 Client가 느끼는 응답 시간이 길어질 것이므로 동일한 Server Process를 여러 개 띄운다든지 하는 방법이 필요할 수 있다.



Service에서 다른 Service를 호출할 수 있도록 허용하는 이유는 이미 작성된 Service를 재사용할 수 있게하며, 만일 Service들이 서로 다른 Machine에 분산되어 있다면 일반적인 함수 호출이 불가능하기 때문이다. 그러나 만일 Service들이 같은 Machine에 있게되면 Service호출 방식을 통한 코드의 재사용은 일반적인 함수 호출에 비해 비용이 너무 많이 소요되는 것을 쉽게 알 수 있을 것이다. Service에서 함수를 호출할 때에는 같은 주소 공간을 사용하기 때문에 Stack에 매개변수를 집어넣고 바로 함수로 분기하면 되지만, Service를 호출할 때에는 기본적으로 서로 다른 Process(그러므로 주소 공간이 다르다.)를 가정(실제적으로는 다른 Machine을 가정)하고 발생하는 것이므로 Process간의 통신이 발생한다. 또한 Service에서 다른 Machine에 있는 Service를 호출할 때에도 tpcall()같은 Blocking Call을 사용하는 경우와 tpforward() 같은 Non-Blocking Call을 사용하였을 때와 비교해 보면 Non-Blocking Call을 사용하는 경우가 훨씬 더 효율적임을 알 수 있다.

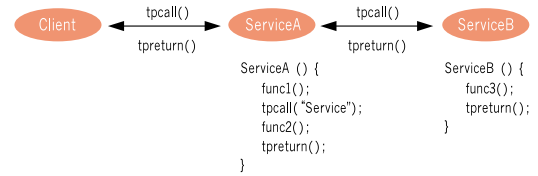




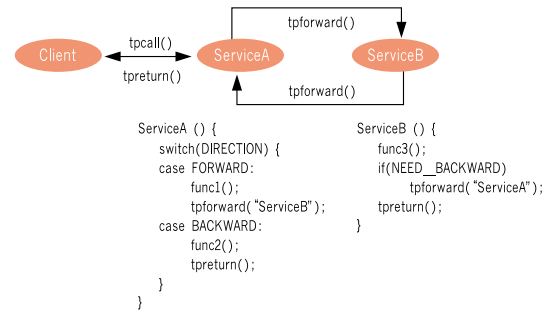
▲ Non-Blocking Call

Non-Blocking Call이 더 효율적인 이유는 첫째 Service가 tpforward()를 호출하면 이 Server Process는 두 번째 Server로부터의 응답을 기다리지 않고 바로 다른 메시지를 처리할 수 있기 때문이다. 즉, Blocking Call에서는 모든 호출이 순차적으로 처리되는데 비해 Non-Blocking Call에서는 이들이 일종의 Pipelining 처리가 되어 병렬로 처리가 된다. Blocking Call에서는 첫째 Service가 두 번째 Service로부터의 응답을 기다렸다가 처리하기 때문에 그 동안 첫째 Server는 다른 메시지를 처리하지 못하기 때문에 자원의 낭비를 가져오게 된다. 또한 그 동안 RM에 특별한 작업을 하지 않더라도 tpreturn()안에서 xa_end()를 호출하기 때문에 XA Session의 종료를 선언할 수 없어서 RM의 Session도 낭비되고 있을 것이다. 만일 Domain간의 호출에서 Blocking Call을 사용한다면 그 심각성은 더할 것이다. 그러므로 Service간의 호출은 일단 최소로 해야하며, 또한 반드시 필요한 경우라도 가능한 Non-Blocking Call을 사용할 수 있게끔 프로그래밍을 해야 할 것이다.

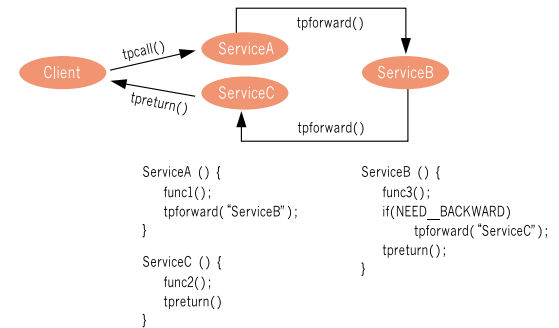
아래의 그림은 Service A에서 Service B를 호출할 때 Blocking Call을 사용하는 경우와 Non-Blocking Call을 사용하는 경우에서 Throughput과 Response Time을 비교한 것이다. func1(), func2(), func3()가 수행되는데 각각 1초가 소요된다고 가정하자. 아래의 경우는 ServiceA와 ServiceB가 같은 Machine에 있던 다른 Machine에 있던 큰 차이는 없다. ServiceA로의 호출이 동시에 3개가 발생했다고 가정했을 때의 상황이 아래의 표에 나타나있다.



▲ Blocking Call



▲ Non-Blocking Call(1)



▲ Non-Blocking Call(2)

	순서	Queue	ServiceA	ServiceB		Response Time
Blocking Call	1	0	2	1		3
	2	3	2	1		6
	3	6	2	1		9
TPS = 3 / 9 = 0.33 Average Response Time = (3+6+9)/3 = 6						

Non-Blocking Call(1)	순서	Queue	ServiceA	ServiceB	ServiceA	Response Time
	1	0+1	1	1	1	4
	2	1+1	1	1	1	5
	3	2+1	1	1	1	6
TPS=3 / 6 = 0.50 Average Response Time = (4+5+6)/3 = 5						
Non-Blocking Call(2)	순서	Queue	ServiceA	ServiceB	ServiceA	Response Time
	1	0	1	1	1	3
	2	1	1	1	1	4
	3	2	1	1	1	5
TPS = 3 / 5 = 0.60 Average Response Time = (3+4+5) / 3 = 4						

Non-Blocking Call(1)에서 Queue에 대기하는 시간이 위와 같이 표시된 이유는 ServiceA가 두번 호출되기 때문이다. 위의 표에서 보드시 TPS나 Response Time에서 모두 Non-Blocking Call이 월등한 성능을 보이는 것을 알 수 있다. 만일 동시에 더 많은 호출이 발생한다면 그 차이는 더욱 더 커질 것이다. 위의 표는 부하가 많이 걸리는 시간대의 경우를 나타내게 될 것이다. 그러나 문제는 차츰 부하가 줄어 들더라도 Queue에 메시지가 남아있는 상황에서는 사용자가 느끼는 Response Time은 크게 줄어들지 않을 거라는 점이다. 가령 Blocking Call에서 처음에 3개의 호출이 동시에 발생하고 이후에 3초에 1개씩 호출이 연속적으로 발생한다고 가정하면 Queue에는 항상 3개의 메시지가 남아있게 되므로 사용자가 느끼는 Response Time은 항상 9초가 될 것이다. Non-Blocking Call에서는 1초에 1개 이상으로 호출이 발생하지 않는 이상 중국에는 사용자가 느끼는 Response Time은 최소값인 3에 다다를 것이다. Non-Blocking Call(1)에서는 ServiceB에서 ServiceA로 다시 호출되는 과정에서 Queue에 대기하는 시간이 발생할 수 있지만 Non-Blocking Call(2)에서는 이러한 대기시간이 발생하지 않는다.

다음의 표는 각각의 방식에서 Queue에 메시지가 남아있지 않을 정도로 호출이 발생한다고 가정했을 경우에 각각의 방식이 기록할 수 있는 최대의 TPS를 나타낸 것이다. 이와 같은 Blocking Call과 Non-Blocking Call간의 차이는 하나의 Transaction안에 포함된 Service의 개수가 많아질수록 그 차이는 더욱 더 커질 것이다.

Blocking Call	TPS = 0.33 Average Response Time = 3
Non-Blocking Call	TPS = 1.00 Average Response Time = 3

Service간의 호출에서 이러한 Blocking Call이 불가피하게 되는 경우는 Client가 아닌 Service에서 Global Transaction을 시작하는 경우일 것이다. Global Transaction의 완료 요청은 반드시

Global Transaction을 기동한 Process가 하여야 하기 때문이다. 그러므로 Global Transaction의 시작은 반드시 Client에서 하도록 하여 이러한 Blocking Call을 방지하여야 할 것이다.

Service의 재사용 문제에 있어서, Service가 서로 다른 Machine에 있는 경우에는 어쩔 수 없겠지만 Service들이 같은 Machine에 있는 경우에는 Service가 제공하는 Logic을 함수로 만들어 재사용하는 것이 바람직하다. 즉, Service를 재사용하는 것이 아니라 함수를 재사용하는 것이 바람직하다. Service에서는 이러한 함수들을 호출한 다음 바로 Client에 응답을 주거나 다른 Machine의 Service로 Non-Blocking Call을 사용하는 것이 바람직하다. Service에서 함수를 호출해 사용하는 것은 Stack에 변수를 집어넣고 바로 함수로 분기하면 되지만 Service를 호출할 때는 앞에서 보였듯이 훨씬 더 복잡한 경로를 지나게 되기 때문이다.

결론

1. 같은 Machine내에서는(최소한 같은 GROUP내에서는) Service에서 Service를 호출하는 형태의 Transaction이 발생하지 않도록 Service를 구성하여야 한다. 이를 위해서는 단위 업무 기능을 함수로 구축하고 Service에서 이들 함수를 차례로 호출하도록 한다. 그래서 Service를 마치함수 호출하듯이 사용하는 형태의 프로그램이 나오지 않도록 한다.

2. 만일 다른 Machine의 Service를 호출할 필요가 있을 때에는 tpcall()같은 Blocking Call 대신에 tpforward()같은 Non-Blocking Call을 사용하여 Server Process가 필요 없이 대기하고 있는 상황을 방지한다. 다른 Service를 호출하는 Service가 호출의 결과를 필요로 하는 곳에서는 Service를 분할하는 것이 좋다. Non-Blocking Call을 사용하는 것은 CPU에서 Pipeline을 사용하는 것과 같은 형태의 기법이다. 그래서 Non-Blocking Call을 사용할 때에도 이러한 Pipeline의 효과를 극대화할 수 있는 방법을 사용하여야 할 것이다.

3. Blocking Call이 발생하는 것을 줄이기 위해 Global Transaction의 기동(tpbegin())과 완료(tpcommit())은 반드시 Client에서 호출하도록 한다. Service에서 Global Transaction을 기동하고 완료하는 경우에는완료의 성공메시지가 Service로 반환된 후에 Service가 Client로 메시지를 반환하기 전에 장애가 발생하면 Global Transaction은 정상적으로 완료되었지만 Client에서는 이 사실을 알 수 없는 상황이 발생할 수 있다. 그러므로 원칙적으로 Global Transaction의 기동과 완료를 Client에서하여야 한다.