Remote Procedure Call Network Library Ver.0.1

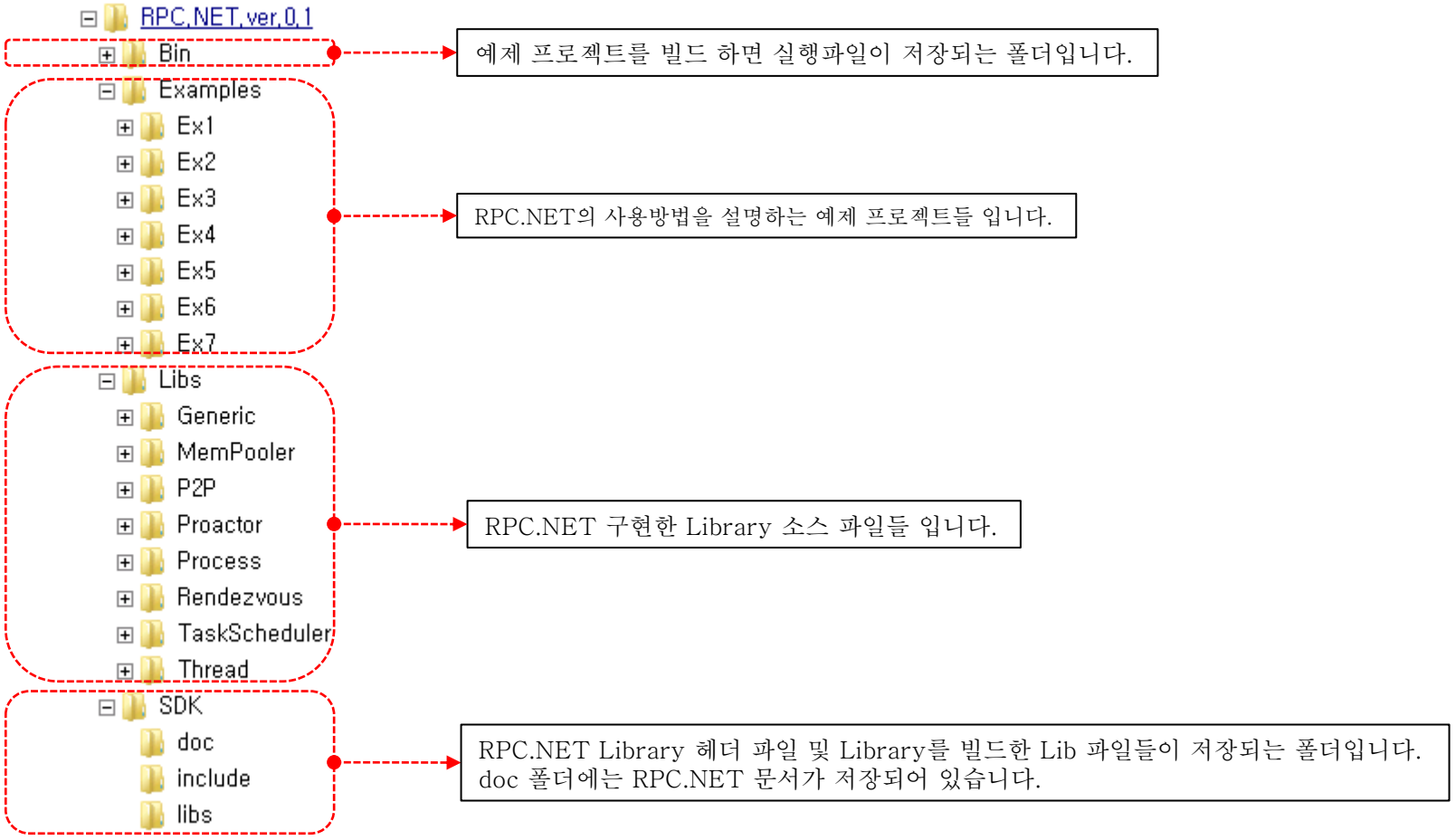
목차

1. RPC.NET 소개
- 솔루션 구조
- 컴파일 옵션 설정
- 빌드 절차
- 라이브러리 종속성

2. RPC.NET 사용방법
- 컴파일 환경 설정
- 비동기 프로시져 호출
- TCP/IP, UDP 네트웍
- 원격 프로시져 호출
- P2P 네트웍
- 무잠금 메모리풀

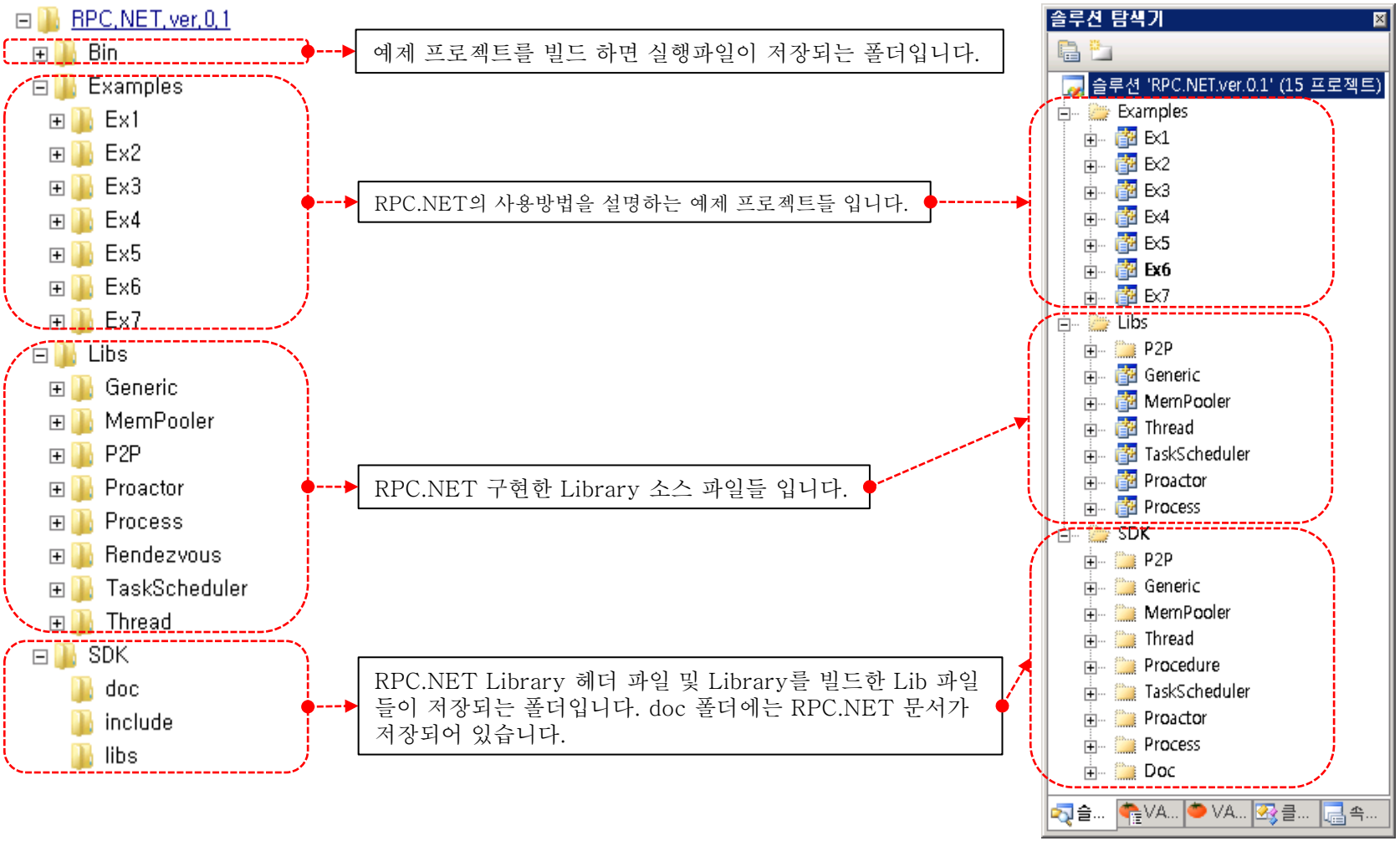
솔루션 구조

배포된 RPC.NET.ver.0.1.zip 파일의 압축을 해제 하면 라이브러리 소스와 예제 그리고 문서가 포함된 폴더가 생성됩니다.



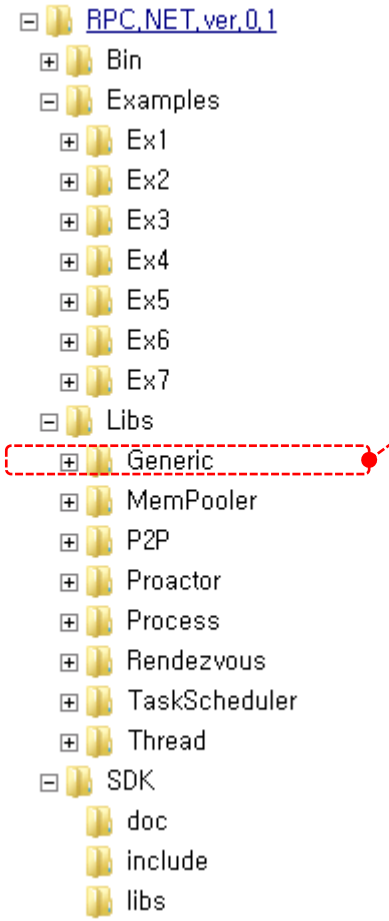
솔루션 구조

라이브러리를 빌드 하기 위해 RPC.NET.ver.0.1 폴더 밑의 **RPC.NET.ver.0.1.sln** 솔루션 파일을 오픈 하시면 좌측의 폴더 구조와 유사한 솔루션 탐색기를 보실 수 있습니다.



컴파일 옵션 설정

RPC.NET 라이브러리의 Compile.h 헤더파일에는 라이브러리에서 사용하는 중요한 상수들을 정의하고 있습니다. 이 상수 값들을 변경하여 RPC.NET의 실행 환경을 변경할 수 있습니다. 정의된 상수들의 세부 설명은 아래 도식을 참조하시기 바랍니다.

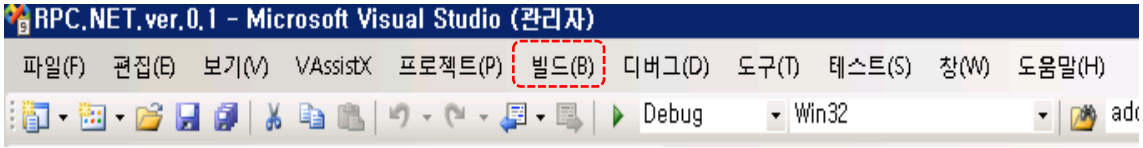


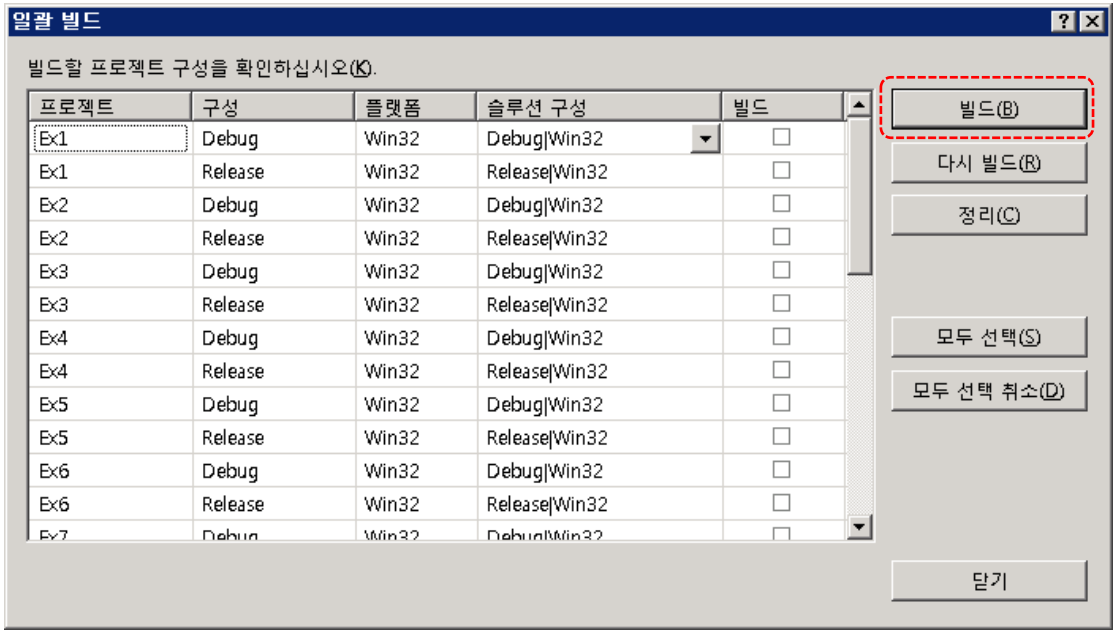
Compile.h 파일 – 만약 아래 상수 값을 변경하면 전체 라이브러리를 다시 빌드 하셔야 됩니다.

```
// TLS 에 생성되는 에러 메시지 버퍼크기
#define ERROR_MESSAGE_BUFFER_SIZE                2048

// P2P 패킷 응답시간을 담아 놓는 Array 의 크기
#define MAX_LATENCY                              10
// P2P reliable packet 을 재전송하는 최소 시간
#define MIN_PACKET_RESEND_LATENCY                100
// P2P 패킷을 한번에 수신할 수 있는 시간(msec)
#define MAX_PACKET_RECEIVE_TM                    50
// 홀펀칭 패킷(Sync)을 전송하는 최대 횟수
#define MAX_SYNC_CNT                             3
```

빌드 절차

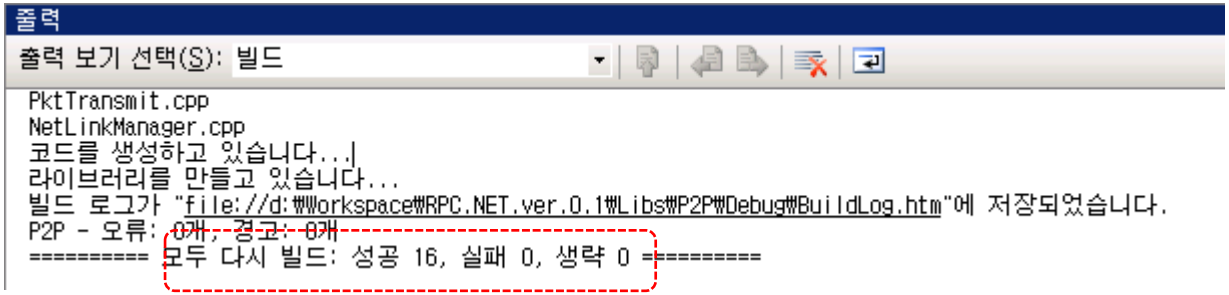
RPC.NET 라이브러리를 빌드 하기 위해서 빌드메뉴의 일괄빌드를 선택한 후 예제 프로젝트들을 – 프로젝트 명이 ex로 시작하는 프로젝트 – 제외한 나머지를 모두 선택하신 후에 빌드 버튼을 클릭 합니다. 참고로 RPC.NET 솔루션에는 win32 debug/release 구성만이 있으므로 x64 라이브러리를 빌드 하시려면 x64 구성을 직접 추가하셔야 합니다.

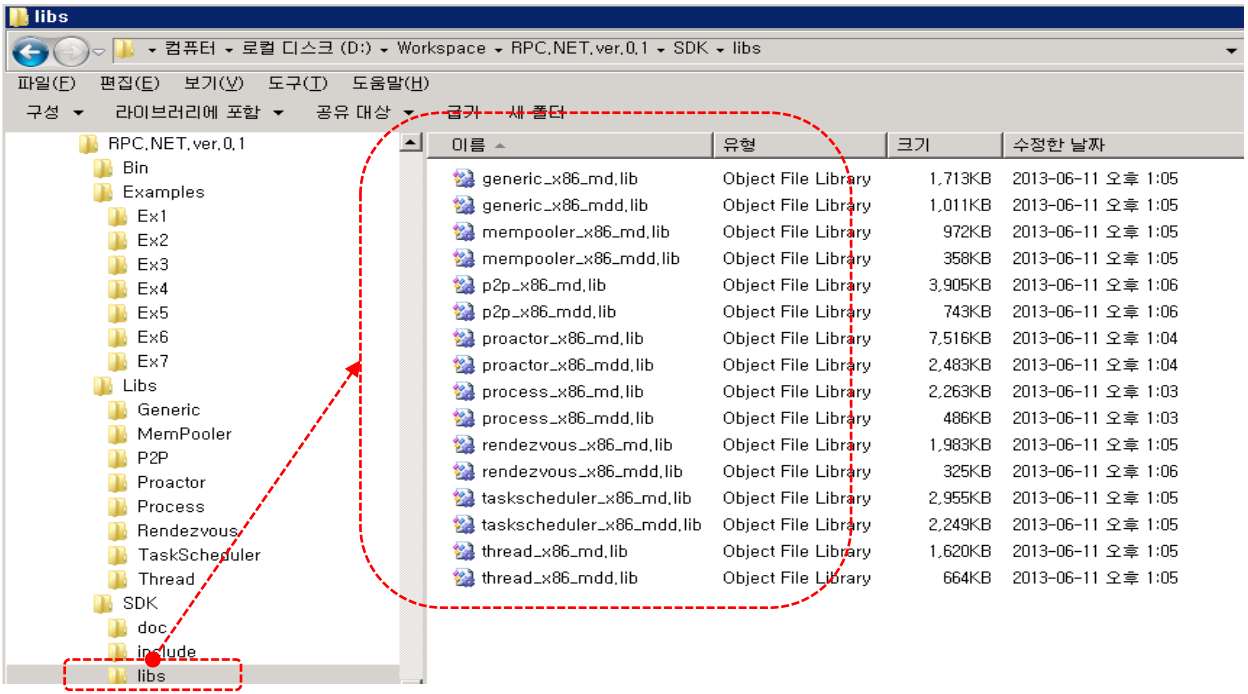




빌드 절차

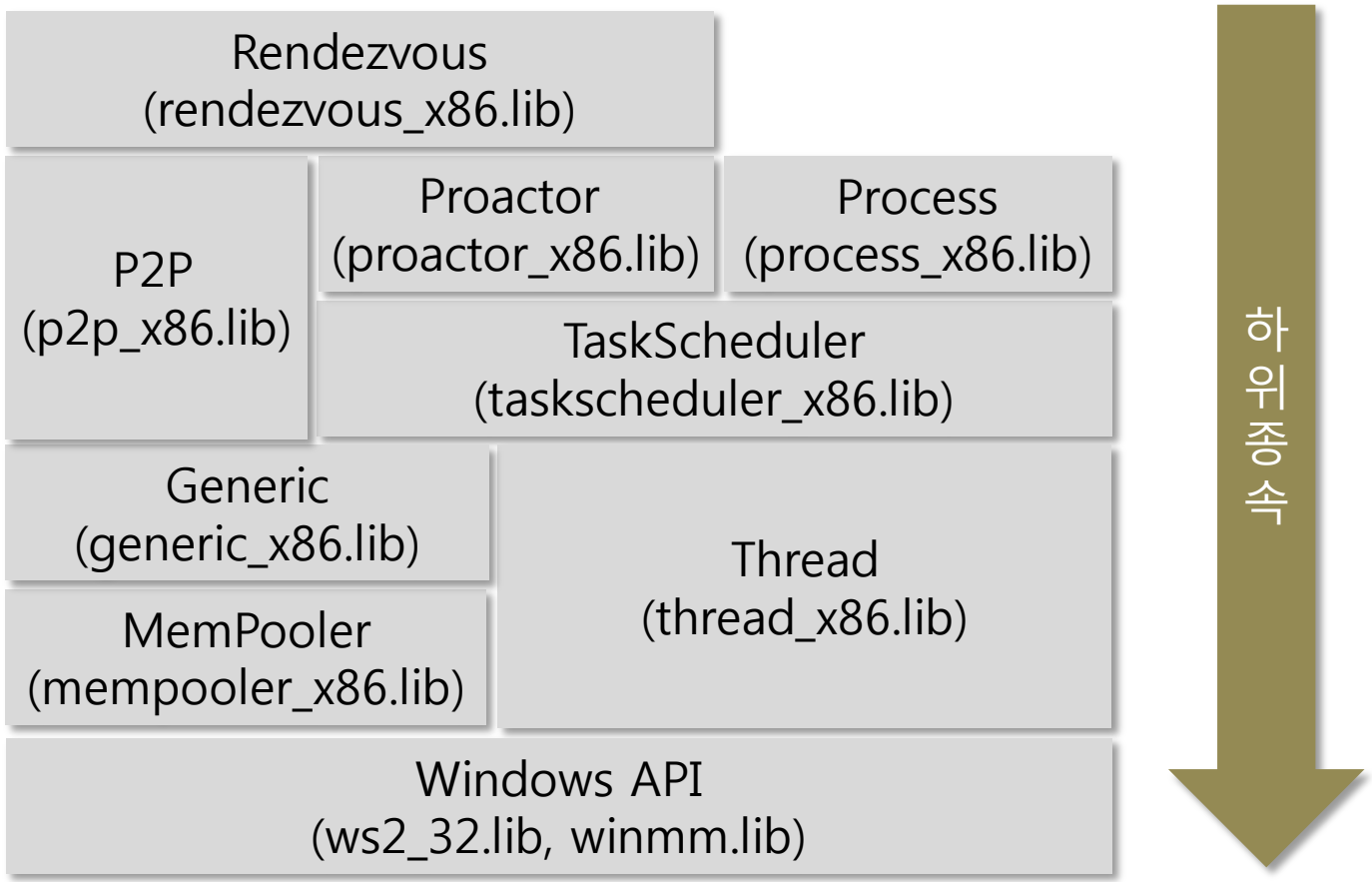
컴파일이 정상적으로 완료되면 빌드 성공 메시지와 RPC.NET.ver.0.1\SDK\libs 폴더 밑에 라이브러리 파일 18개가 생성된 것을 확인 하실 수 있습니다.





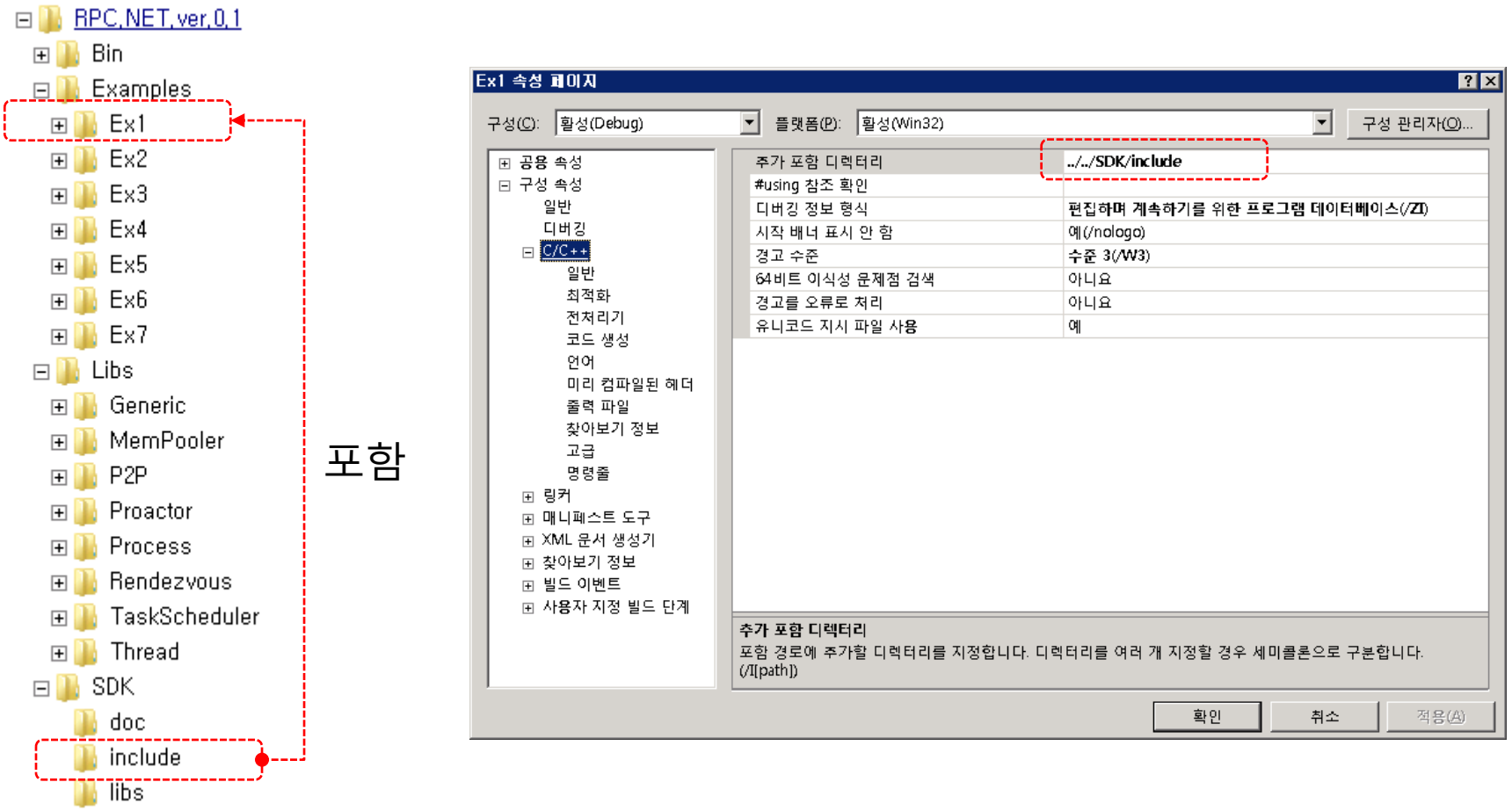
라이브러리 종속성

RPC.NET 을 구성하는 라이브러리들은 아래의 도식과 같이 하위 종속성을 가지고 있습니다. 도식에서 상위 라이브러리는 자신의 바로 아래에 위치한 라이브러리들에 종속됩니다. 예를 들어 Proactor 라이브러리는 TaskScheduler, Generic, MemPooler, Thread 라이브러리에 종속됩니다.



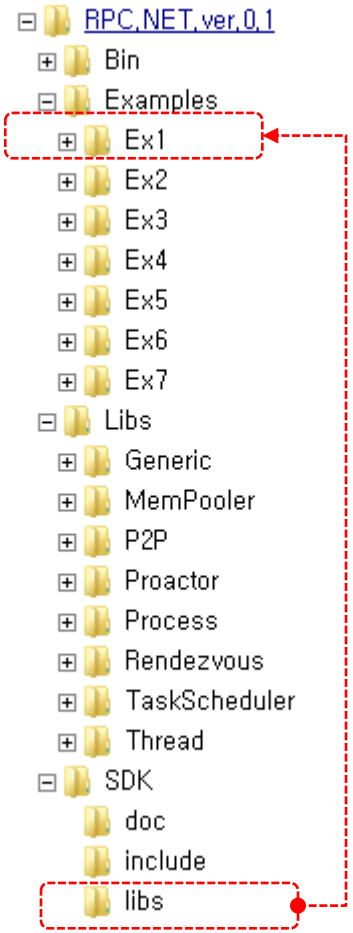
컴파일 환경 설정

RPC.NET 을 자신의 프로젝트에 사용하기 위해서 속성페이지의 컴파일 옵션 과 링크 옵션을 설정해야 합니다. 먼저 해야 할 일은 속성페이지의 C/C++ 항목을 선택한 후 “추가 포함 디렉터리” 항목에 RPC.NET 라이브러리의 헤더파일들이 포함되어 있는 RPC.NET.ver.0.1\SDK\include 를 입력해야 합니다.
아래의 그림은 Ex1 예제의 “추가 포함 디렉터리”를 ../../SDK/include 로 설정한 예를 보여줍니다.

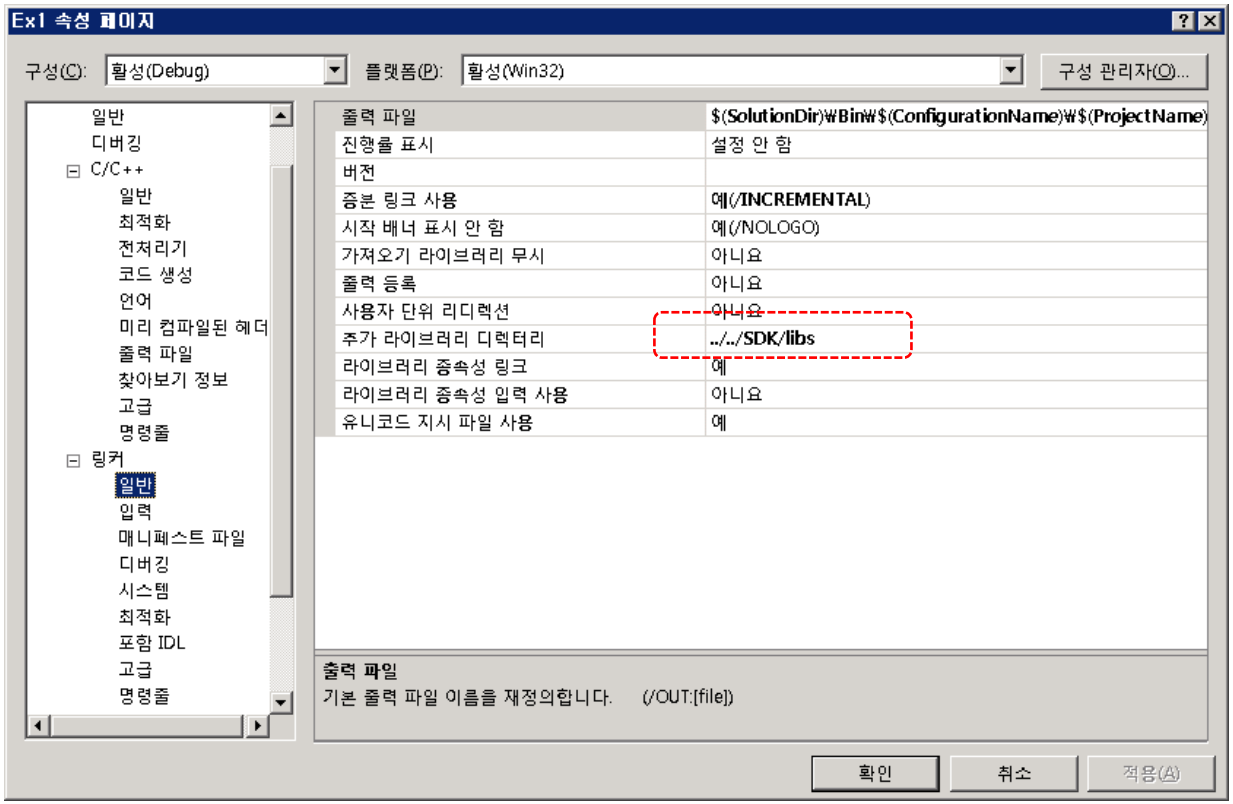


컴파일 환경 설정

한가지 더 해야 할 일은 VS에게 RPC.NET 라이브러리의 위치를 알려주기 위해 링커 옵션을 설정하는 것입니다. 속성페이지에서 링커 항목을 선택한 후 “추가 라이브러리 디렉터리”에 RPC.NET.ver.0.1\SDK\libs 를 입력해야 합니다.
아래의 그림은 Ex1 예제의 “추가 포함 디렉터리”를 ../../SDK/libs 로 설정한 예를 보여줍니다.



포함



컴파일 환경 설정

RPC.NET 라이브러리를 프로젝트에 링크하기 위해 추가종속성 항목을 설정 하실 필요는 없습니다. RPC.NET 헤더파일에는 컴파일 지시어(#pragma comment)를 사용하여 링크할 라이브러리명을 명시하고 있기 때문에 단지 헤더 파일을 프로젝트의 소스 파일에 포함시켜는 것 만으로 라이브러리를 링크할 수 있습니다. 또한 종속성에 따라 상위 헤더파일에는 하위 헤더파일을 포함하고 있으므로 각각의 헤더파일들을 프로젝트에 포함하지 않고 단지 상위 헤더파일 한 개만 포함하시면, 나머지 하위 헤더파일들이 같이 포함됩니다. 예를 들어 Proactor 라이브러리를 사용할 경우 Proactor.h 를 포함하는 것만으로 나머지 MemPooler.h, Generic.h, TaskScheduler.h, ThreadEx.h 도 같이 포함됩니다.

헤더파일	라이브러리 설명
MemoryEx.h	무잠금 메모리풀 및 데이터 저장 버퍼 클래스
Generic.h	공통 헤더 및 유틸리티(문자열, 시간 함수, MD5 해쉬)
ThreadEx.h	쓰레드 오브젝트 생성, 실행, 종료
TaskScheduler.h	작업 및 타이머 실행, 비동기 프로시져 실행
Proactor.h	TCP/IP, UDP 네트웍, RPC(Remote Procedure Call) 실행
ProcessEx.h	프로세스 실행, 종료 및 상태 감시
P2P.h	P2P 클라이언트(홀펀칭, P2P 패킷 송수신)
Rendezvous.h	P2P 랑데부, P2P 패킷 릴레이

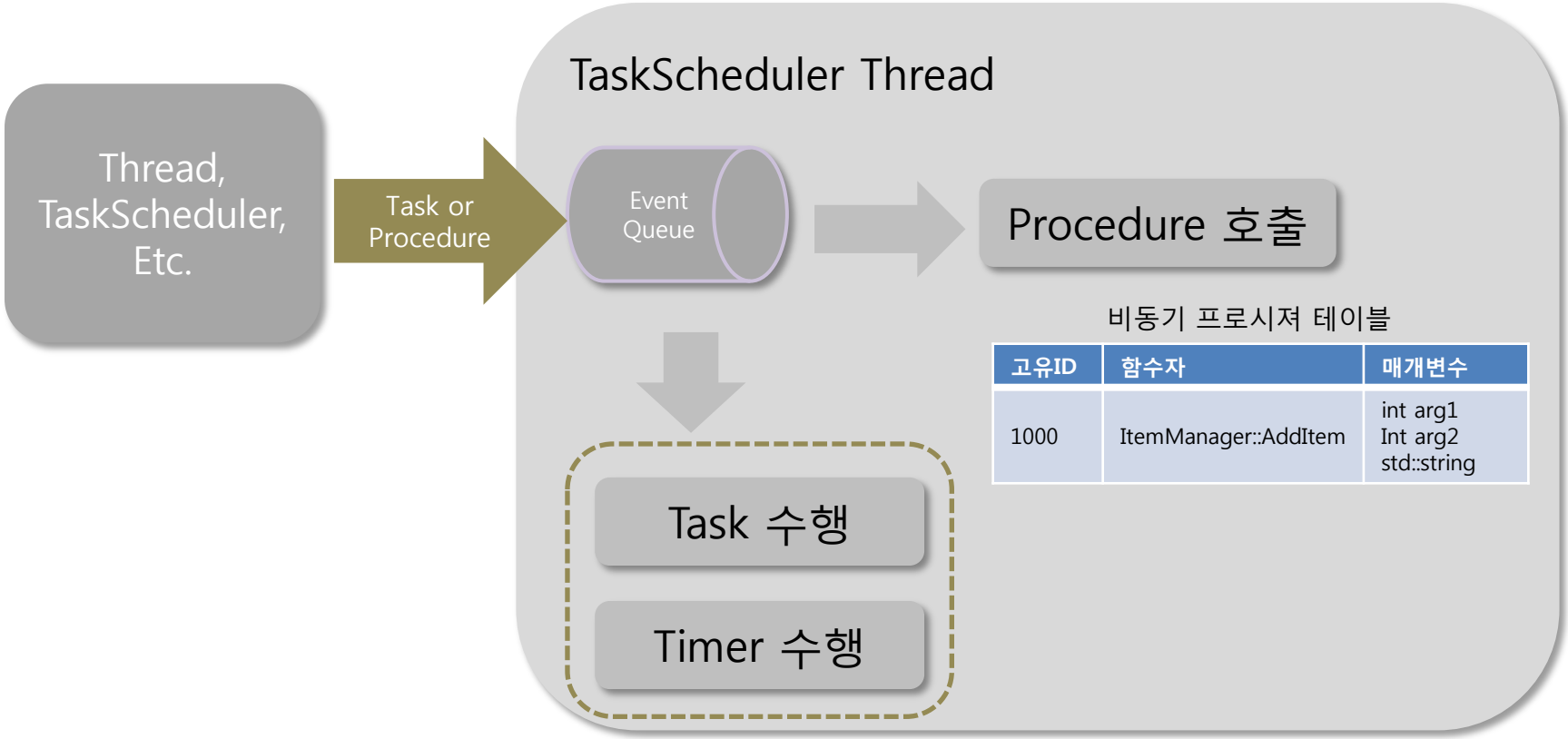
컴파일 환경 설정

아래는 무잠금 메모리 풀 라이브러리를 프로젝트에 링크시키는 방법을 보여줍니다.

```
// 무잠금 메모리풀을 사용
#include <MemoryEx.h>

class MyObj
{
public:
    MyObj() {}

    // 디스트럭터 앞에는 반드시 virtual 을 붙여야 한다.
    virtual ~MyObj() {}
    // 무잠금 메모리 풀에서  오브젝트를 생성하려면 정의해야 합니다.
    AllocatorDecl();
};

int _tmain(int argc, _TCHAR* argv[])
{
    // MyObj 를 new 연산자로 할당하면 무잠금 메모리풀에서 할당한다.
    MyObj* pObj = new MyObj;

    // 사용이 끝난 MyObj를 삭제하면 메모리가 무잠금 메모리풀에 반환된다.
    if ( pObj ) delete pObj;

    return 0;
}
```

비동기 프로시져 호출

RPC.NET의 태스크 스케줄러는 Application내에서 비동기적으로 호출되는 작업과 타이머을 관리합니다. 태스크 스케줄러가 실행되면 1개의 쓰레드가 생성되는데 쓰레드는 작업과 타이머 목록 리스트를 일정한 간격으로 순회하며 호출시간이 만료된 작업이나 타이머의 콜백 함수를 호출합니다. 또한 태스크 스케줄러는 외부에서 비동기 프로시져의 실행을 요청하는 작업이벤트를 받으면 비동기 프로시져 테이블에 등록된 비동기 프로시져를 실행하는 인터페이스를 제공합니다. 아래 도식을 보면 외부의 쓰레드나 태스크스케줄러에서 비동기 프로시져를 실행하라고 요청하는 모습을 보여줍니다.

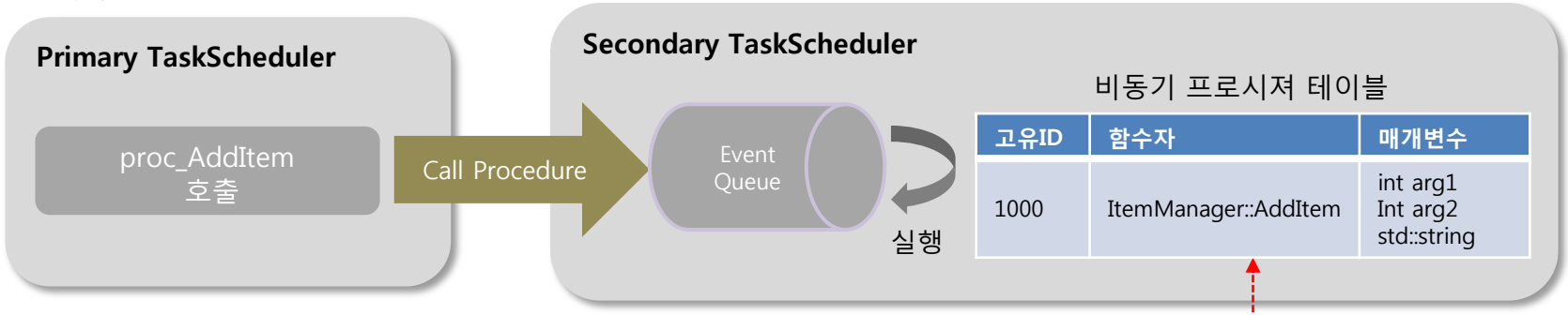


비동기 프로시져 테이블

고유ID	함수자	매개변수
1000	ItemManager::AddItem	int arg1 Int arg2 std::string

비동기 프로시져 호출

비동기 프로시져에 대한 이해를 돕기 위해서 간단한 예제를 만들어 보겠습니다. 아래는 2개의 태스크스케줄러간에 비동기 프로시져를 호출하는 방법을 보여주는데, 먼저 두 번째 태스크 스케줄러에 ItemManager::AddItem 를 비동기 프로시져를 등록하고 첫 번째 태스크 스케줄러에서 두 번째 태스크 스케줄러에 게 ItemManager::AddItem 함수를 실행하도록 요청하는 예제를 구현하겠습니다.

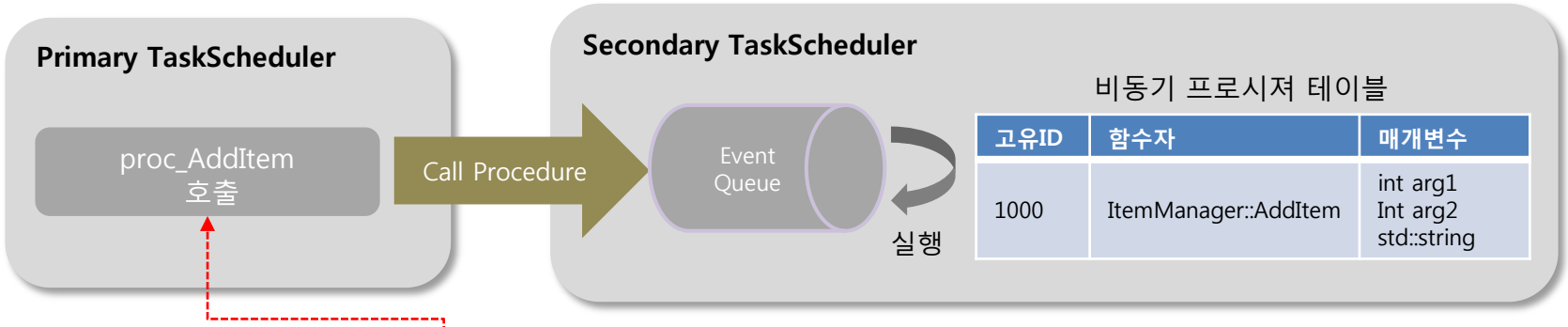



```
class ItemManager
{
public:
    ItemManager()
    {
        EXPORT_PROC_V3( 1, 1000, this, &ItemManager ::AddItem,
                        int, int, const std::string& );
    }
    virtual ~ItemManager() {}

    // 비동기 프로시져로 등록할 함수는 반드시 bool 함수명( IRef*, 매개변수.. ) 형태
    bool AddItem( IRef* pObj, int iPlayerID, int iItemID, const std::string& sItemName )
    {
        printf( __FUNCTION__": Thread %d\n", ::GetCurrentThreadId() );
        return true;
    }
};
```

EXPORT_PROC_V3 매크로를 사용하여 ItemManager::AddItem 함수를 고유ID 1000 번으로 비동기 프로시져 테이블에 등록합니다.

비동기 프로시져 호출

PlayerManager 클래스에 서 ItemManager::AddItem 비동기 프로시져를 호출을 요청할 수 있도록 IMPORT_PROC 매크로를 사용하여 비동기 프로시져 호출함수를 정의합니다.




```
class PlayerManager
{
public:
    PlayerManager()
    {
    }
    virtual ~ PlayerManager()  {}

    IMPORT_PROC_V3( 1000, AddItem, int, int, const std::string&, 1024 );

    void EventTick()
    {
        printf( __FUNCTION__": Thread %d\n", ::GetCurrentThreadId() );

        proc_AddItem ( NULL, 1, i++,  "big hat"  );

    }
};
```

IMPORT_PROC_V3 매크로를 사용하여 ItemManager::AddItem 비동기 프로시져를 호출할 함수를 PlayerManager의 맴버함수로 정의합니다.

ItemManager::AddItem 비동기 프로시져를 2번 태스크 스케줄러에게 실행 하도록 요청합니다. 함수명 앞에 proc_ 를 붙여야 합니다.

비동기 프로시져 호출

ItemManager::AddItem 비동기 프로시져를 호출할 준비가 완료 됐으므로 이제 태스크 스케줄러를 생성하고 1초에 한번씩 비동기 프로시져가 호출되도록 구현해 보겠습니다.

```
#include <Errorcodes.h>
#include <TaskScheduler.h>

int _tmain(int argc, _TCHAR* argv[])
{
   sc1 = CreateTaskScheduler( 1, 50, 1 );
   if ( sc2 == NULL && GetLastError() != NOERROR )
   {
      printf( "error: %s\n", GetLastErrorMessage( GetLastError() ) );
      return 1;
   }

   sc2 = CreateTaskScheduler( 2, 50, 2 );
   if ( sc2 == NULL && GetLastError() != NOERROR )
   {
      printf( "error: %s\n", GetLastErrorMessage( GetLastError() ) );
      return 1;
   }

   ItemManager  inst1;
   PlayerManager inst2;

   CreateTask( sc1, &inst1, &PlayerManager::EventTick, 1000,  "" );

   // 생성한 태스크 스케줄러들을 실행합니다.
   StartTaskScheduler( TS_PRIMARY );
   StartTaskScheduler( TS_SECONDARY );

   getchar();
   // 모든 태스크 스케줄러를 종료합니다.
   StopTaskScheduler( 0 );
   return 0;
}
```

태스크 스케줄러 2개를 생성했습니다.

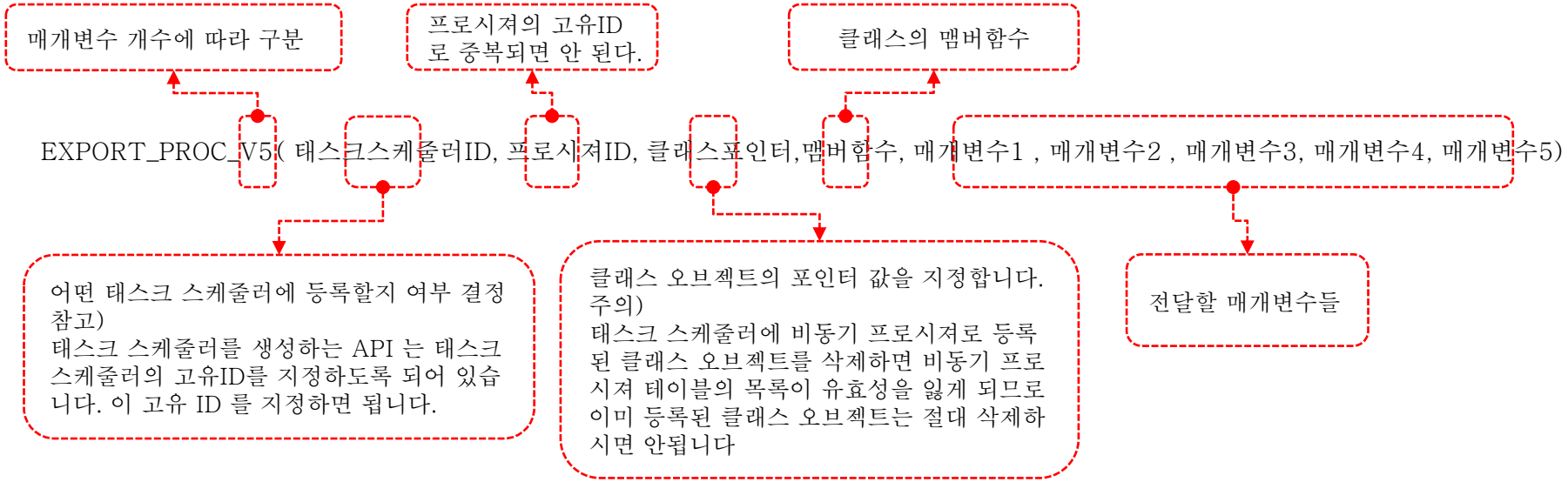
1000ms 간격으로 PlayerManager::EventTick 이 호출 되도록 1번 태스크 스케줄러에 등록합니다.

비동기 프로시져 호출

태스크 스케줄러에 비동기 프로시져를 등록하기 위해서는 EXPORT_PROC 로 시작하는 6 가지 매크로를 사용해야 합니다. 각각의 등록 매크로들은 비동기 프로시져에게 몇 개의 매개변수를 전달할지에 따라 매크로명의 뒤에 V0 ~ V5 가 붙어 있을 뿐 기본 기능은 모두 동일합니다. 주의할 점은 태스크 스케줄러에 비동기 프로시져로 등록된 클래스 오브젝트를 삭제하면 비동기 프로시져 테이블 목록이 유효성을 잃게 되므로 이미 등록된 클래스 오브젝트는 절대 삭제하시면 안됩니다. 아래는 등록 매크로의 원형과 비동기 프로시져로 등록할 함수의 원형을 보여줍니다.

- **EXPORT_PROC_V(태스크스케줄러ID, 프로시져ID, 클래스포인터, 맴버함수, 매개변수자료형들….)**
- **bool 함수명(IRef* , 매개변수들)**

매개변수	설명
태스크스케줄러 ID	비동기 프로시져를 어떤 태스크 스케줄러에서 등록하여 실행할 것인지 설정 합니다.
프로시져 ID	비동기 프로시져 테이블에 검색 키 값으로 사용되는 고유한 ID 입니다. IMPORT_PROC 매크로의 프로시져 ID 와 동일하게 매칭되게 정의한 후에 IMPORT_PROC 매크로에 정의한 함수이름을 호출하면 EXPORT_PROC 로 등록한 비동기 프로시져가 호출됩니다.
클래스포인터	비동기 프로시져로 등록할 맴버함수를 가지고 있는 클래스 오브젝트의 포인터 값입니다.
맴버함수포인터	비동기 프로시져로 등록할 맴버함수의 포인터 값입니다.

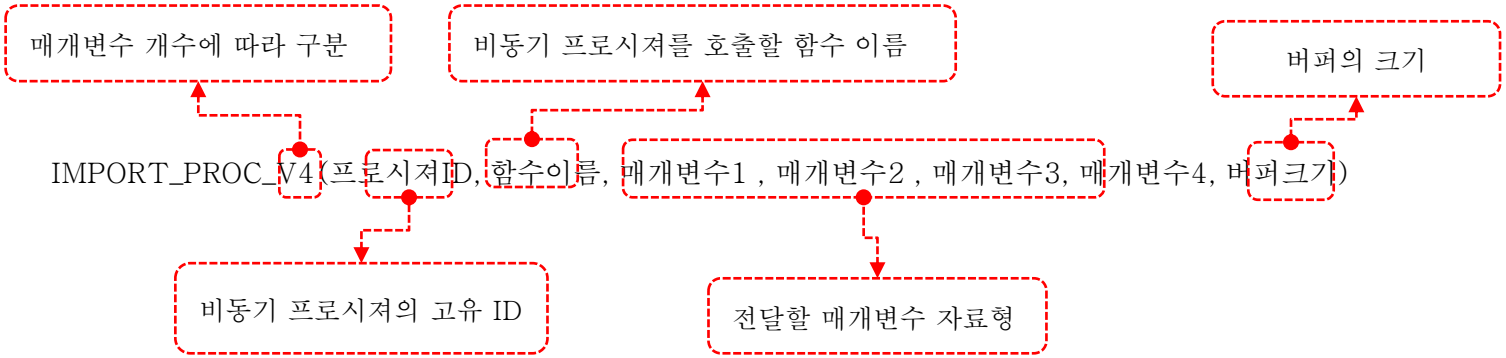


비동기 프로시져 호출

태스크 스케줄러의 비동기 프로시져를 호출 하기 위해서는 IMPORT_PROC 로 시작하는 6 가지 매크로를 사용하여 비동기 프로시져에 대응하는 함수를 선언해야 합니다. IMPORT_PROC 매크로로 선언된 함수를 호출하면 태스크 스케줄러에게 비동기 프로시져를 실행하라는 작업이벤트를 전달합니다. 한가지 주의할 점은 비동기 프로시져를 호출할 때는 함수이름 앞에 **proc_** 를 붙여 호출해야 합니다.

- **IMPORT_PROC_V(비동기프로시져ID, 함수이름, 매개변수자료형들.., 버퍼크기)**
- **bool proc_함수이름(IRef*, 매개변수들…)**

매개변수	설명
비동기프로시져 ID	비동기 프로시져 테이블에 검색 키 값으로 사용되는 고유한 ID 입니다. IMPORT_PROC 매크로의 프로시져 ID 와 동일하게 정의한 후에 IMPORT_PROC 매크로에 정의한 함수이름을 호출하면 EXPORT_PROC 로 등록한 비동기 프로시져가 호출됩니다.
함수이름	비동기 프로시져를 호출할 함수이름으로 IMPORT_PROC_V 매크로에 정의한 함수이름 앞에 proc_ 붙여서 호출합니다. 예를 들어 아래와 같이 함수이름 func 으로 정의하면 호출할 때는 proc_func 으로 호출해야 한다. Ex) IMPORT_PROC_V1(1000, func, int, 1024); proc_func(32);
버퍼크기	비동기 프로시져를 호출할 때 생성되는 버퍼의 크기로 프로시져 ID 및 매개변수가 저장될 공간의 크기를 정의한다.

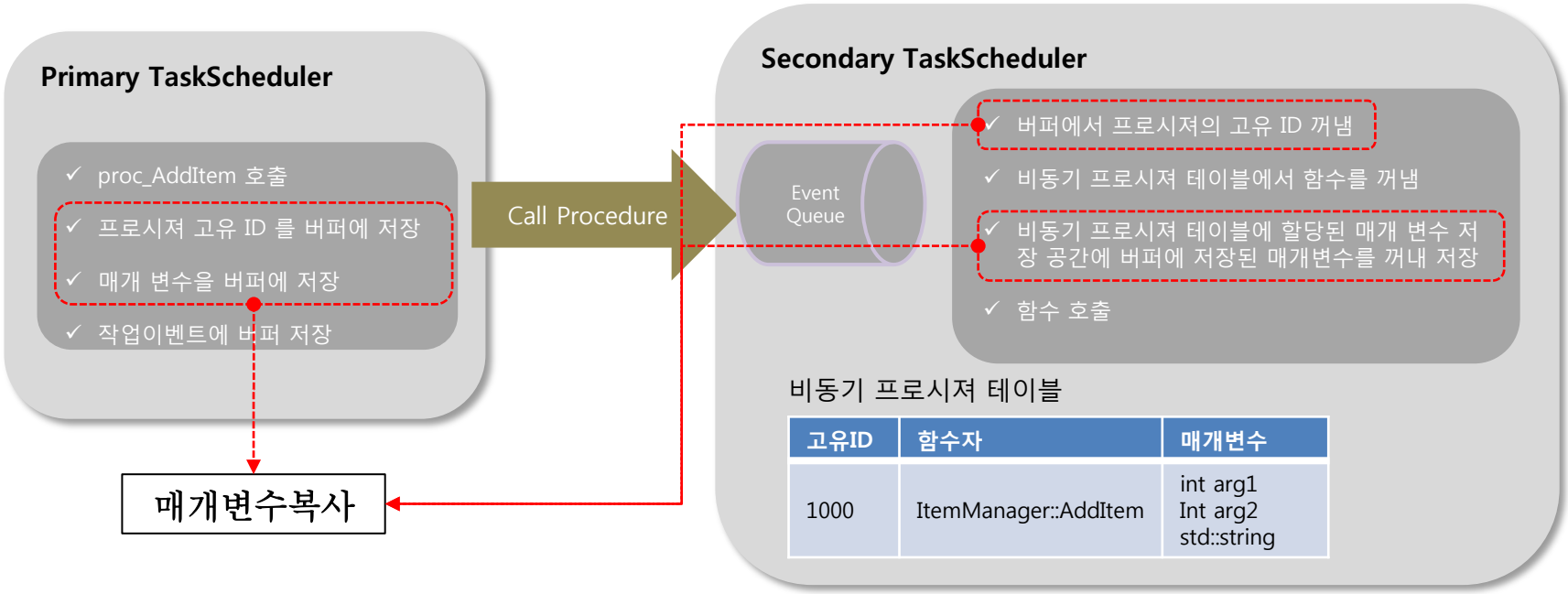


비동기 프로시져 호출

앞 장에서 간단한 예제를 통해 태스크 스케줄러의 비동기 프로시져 호출 메커니즘을 설명 드렸습니다. 이번 장에는 비동기 프로시져를 정의할 때 주의할 사항들에 대해서 설명 드리고자 합니다.

앞에서 설명 드렸듯이 외부에서 비동기 프로시져를 호출하면 작업이벤트를 생성하고 작업이벤트의 메모리 공간에 비동기프로시져ID와 매개변수들을 저장한 후 태스크 스케줄러에게 전달합니다. 태스크 스케줄러는 작업이벤트에서 비동기 프로시져 ID를 꺼내와 비동기 프로시져 테이블에서 매칭되는 비동기 프로시져를 실행합니다.

여기서 주의할 점은 비동기 프로시져에 전달될 매개변수 자료형을 어떻게 선언하느냐에 따라 매개변수의 복사 횟수가 달라져 오버헤드를 증가하거나 감소할 수 있다는 점입니다. 아래의 도식을 보면 1번째 태스크 스케줄러에서 proc_AddItem 을 호출하여 2번째 태스크 스케줄러에 등록된 비동기 프로시져 ItemManager::AddItem 을 호출을 요청합니다. 이 과정에서 매개변수의 복사가 일어납니다.



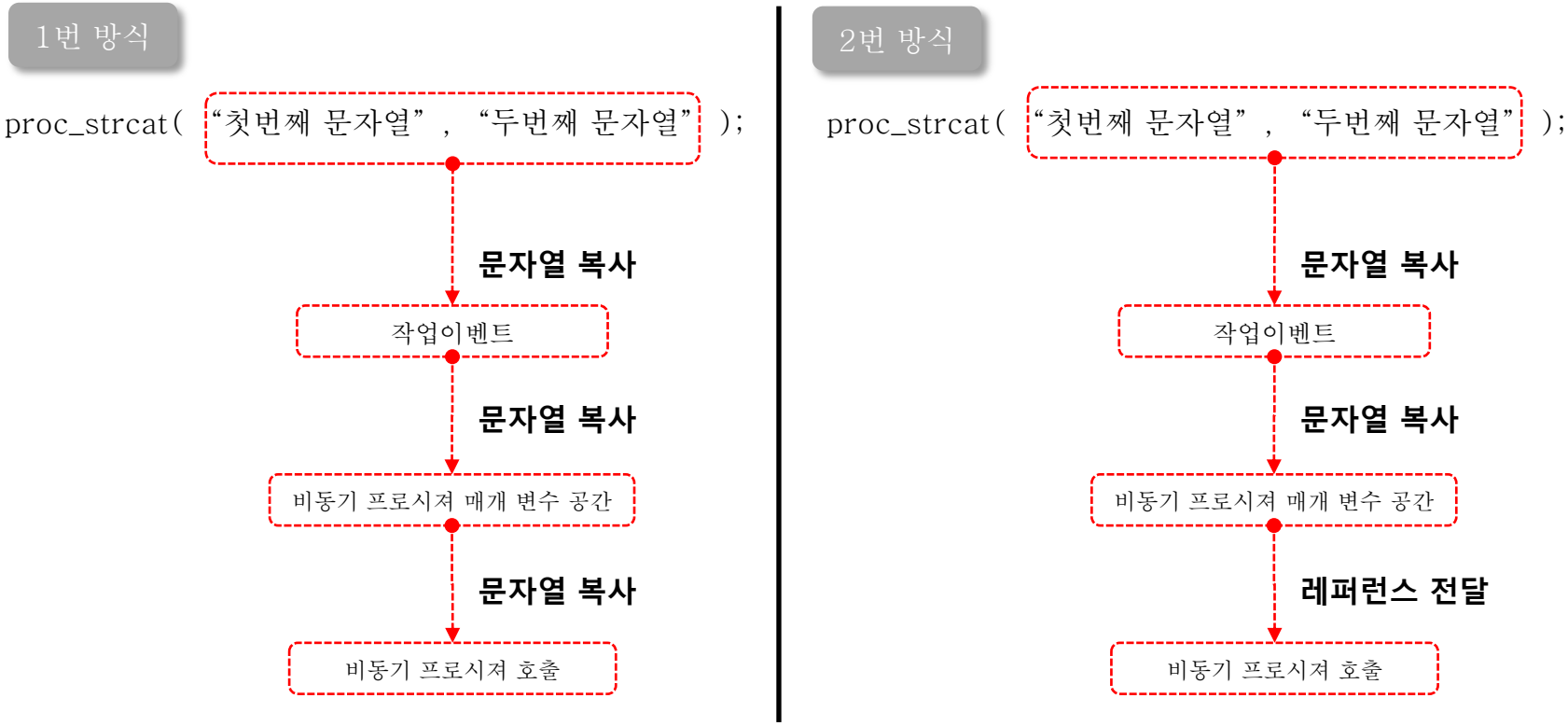
비동기 프로시져 호출

이러한 매개 변수 전달 방식 때문에 비동기 프로시져를 등록할 때 매개 변수들의 자료형을 신중하게 선택하셔야 메모리 복사에 따른 오버헤드를 줄일 수 있습니다.

만약 매개 변수로 입력 받은 두 문자열을 합치는 아래의 두 함수를 비동기 프로시져로 등록한 다고 할 때

1) bool strcat(IRef* pObj, const std::string a, const std::string b);
2) bool strcat(IRef* pObj, const std::string& a, const std::string& b);

2번 방식으로 매개 변수를 선언하는 것이 메모리 복사에 따른 오버헤드를 줄이는 좋은 방법입니다.



비동기 프로시져 호출

태스크 스케줄러API 목록은 아래와 같습니다.

함수명	함수 원형	설명

CreateTaskScheduler	TaskScheduler* CreateTaskScheduler(
	_in unsigned int iTaskSchedulerID,
	_in unsigned long iDuration,
	_in iBindCPU)
	
	Ex)
	TaskScheduler* sc = CreateTaskScheduler(
	1,
	50,
	1);
	if (sc == NULL & GetLastError() != NOERROR)
	{
	printf("error: %s\n" ,
	GetLastErrorMessage(GetLastError()));
	exit(0);
	}
		태스크 스케줄러를 생성합니다.
		-. iTaskSchedulerID
		생성할 태스크 스케줄러의 고유한 ID 값으로 1 ~ 정수최대값을 가집니다. 태스크 스케줄러를 검색하거나 태스크, 타이머, 비동기 프로시져를 등록할 때 어떤 태스크 스케줄러에 등록해야 하는지를 지정하게 되어 있는데 이 값을 사용합니다.
		-. iDuration
		태스크 스케줄러에 등록된 태스크나 타이머의 호출시간이 만료됐는지 조사하는 간격입니다. 설정 값은 ms 단위로 지정해야 합니다.
		-. iBindCPU
		태스크 스케줄러를 생성하면 쓰레드가 1개 생성되는데 이 쓰레드를 어떤 CPU 에서 실행하도록 할 것 인지를 설정합니다.

StartTaskScheduler	bool StartTaskScheduler(_in unsigned int iTaskSchedulerID) Ex) StartTaskScheduler(1);	태스크 스케줄러를 실행한다. -. iTaskSchedulerID 생성할 태스크 스케줄러의 고유한 ID 값으로 1 ~ 정수최대값을 가집니다. 태스크 스케줄러를 검색하거나 태스크, 타이머, 비동기 프로시져를 등록할 때 어떤 태스크 스케줄러에 등록해야 하는지를 지정하며 이 값을 사용합니다. 만약 0 을 지정하면 현재 생성된 모든 태스크 스케줄러를 실행합니다.
CreateTask	int CreateTask(_in unsigned int iTaskSchedulerID, _in const O& o, _in F f, _in unsigned int iDuration) Ex) CreateTask(1, &obj, &MyObj::Func1, 1000);	태스크 스케줄러에 태스크를 등록합니다. -. iTaskSchedulerID 생성할 태스크 스케줄러의 고유한 ID 값으로 1 ~ 정수최대값을 가집니다. -. o 호출할 클래스 객체의 주소값 -. f 호출할 맴버함수의 주소값 -. iDuration 태스크를 호출할 간격

비동기 프로시져 호출

함수명	함수 원형	설명
CreateTimer	int CreateTimer(_in unsigned int iTaskSchedulerID, _in const O& o, _in F f, _in unsigned int iDuration, _in void* pUserData) Ex) CreateTimer(1, &obj, &MyObj::Func1, 1000, NULL);	태스크 스케줄러에 반복 타이머를 등록합니다. -. iTaskSchedulerID 생성할 태스크 스케줄러의 고유한 ID 값으로 1 ~ 정수최대값을 가집니다. -. o 호출할 클래스 객체의 주소값 -. f 호출할 맴버함수의 주소값 -. iDuration 태스크를 호출할 간격 -. pUserData 타이머에 전달할 주소값
StopTaskScheduler	bool StopTaskScheduler(_in unsigned int iTaskSchedulerID,)	태스크 스케줄러에 반복 타이머를 등록합니다. -. iTaskSchedulerID 생성할 태스크 스케줄러의 고유한 ID 값으로 1 ~ 정수최대값을 가집니다. 만약 0 이면 모든 태스크스케줄러를 종료합니다.

비동기 프로시져 호출

EXPORT_PROC 매크로의 정의는 아래와 같습니다.

매크로	선언방법
EXPORT_PROC_V0	EXPORT_PROC_V0(1, 1000, this, &ItemManager::AddItem);
EXPORT_PROC_V1	EXPORT_PROC_V1(1, 1001, this, &ItemManager::AddItem, int);
EXPORT_PROC_V2	EXPORT_PROC_V2(1, 1002, this, &ItemManager::AddItem, int, int);
EXPORT_PROC_V3	EXPORT_PROC_V3(1, 1003, this, &ItemManager::AddItem, int, int, const std::string&);
EXPORT_PROC_V4	EXPORT_PROC_V4(1, 1004, this, &ItemManager::AddItem, int, int, int, float);
EXPORT_PROC_V5	EXPORT_PROC_V5(1, 1005, this, &ItemManager::AddItem, int, int, double, int, float);

IMPORT_PROC 매크로의 정의는 아래와 같습니다.

매크로	선언방법
IMPORT_PROC_V0	IMPORT_PROC_V0(1000, AddItem);
IMPORT_PROC_V1	IMPORT_PROC_V1(1001, AddItem, int, 1024);
IMPORT_PROC_V2	IMPORT_PROC_V2(1002, AddItem, int, int, 1024);
IMPORT_PROC_V3	IMPORT_PROC_V3(1003, AddItem, int, int, const std::string&, 1024);
IMPORT_PROC_V4	IMPORT_PROC_V4(1004, AddItem, int, int, int, float1024);
IMPORT_PROC_V5	IMPORT_PROC_V5(1005, AddItem, int, int, double, int, float, 1024);

비동기 프로시져 호출

비동기 프로시져로 호출할 맴버함수들의 매개변수 자료형은 아래 목록과 같습니다.

구분	자료형	선언방법
문자열	std::string / const std::string&, mem::string / const mem::string&	EXPORT_PROC_V1(1, 1000, this, &TestObj2::Func2, const std::string&);
스칼라	char / unsigned char, short / unsigned short, int / unsigned int, long / unsigned long, __int64 / unsigned __int64, float, double	EXPORT_PROC_V1(1, 1001, this, &TestObj2::Func3, int);
포인터	ptr_t 포인터값을 전달하기 위한 구조체 ptr_t obj; int a = 32; obj = &a;	EXPORT_PROC_V1(1, 1002, this, &TestObj2::Func2, const ptr_t&);

구조체	Buffer 의 전역 연산자 << 와 >> 를 구현하여 구조체 전달할 수 있습니다.
	struct custom
	{
	int a1;
	std::string s1;
	};
	
	저장 연산자 구현
	Buffer& operator<<(Buffer& pk, const custom& r)
	{
	pk << r.a1 << r.s1;
	return pk;
	}
	
	Buffer& operator>>(Buffer& pk, const custom& r)
	{
	pk >> r.a1 >> r.s1;
	return pk;
	}
		EXPORT_PROC_V1(1, 1003, this, &TestObj2::Func3, const custom&);

TCP/IP, UDP 네트워

일반적인 네트워 프로그래밍에서 TCP/IP 소켓을 사용하는 방법은 서버에서는 소켓을 생성하여 클라이언트의 접속을 받아들일 준비를 하고, 클라이언트에서는 소켓을 생성하여 서버에 연결한 후 데이트를 송수신 하도록 구현합니다.

```
struct sockaddr_in srvaddr;

memset( &srvaddr, 0x00, sizeof(srvaddr) );
srvaddr.sin_family = AF_INET;
srvaddr.sin_addr.s_addr = inet_addr( "127.0.0.1" );
srvaddr.sin_port = htons( 50000 );

fd = socket(AF_INET, SOCK_STREAM, 0);

conect( fd, (struct sockaddr*)&srvaddr, sizeof(srvaddr));
```

연결

```
struct sockaddr_in srvaddr, cliaddr;

memset( &srvaddr, 0x00, sizeof(srvaddr) );
srvaddr.sin_family = AF_INET;
srvaddr.sin_addr.s_addr = inet_addr( "127.0.0.1" );
srvaddr.sin_port = htons( 50000 );

fd = socket(AF_INET, SOCK_STREAM, 0);

bind(fd, (struct sockaddr *)&srvaddr, sizeof(srvaddr));
listen(fd, 128);
int clilen = sizeof(cliaddr);
newfd = accept(fd, (struct sockaddr*)&cliaddr, &clilen);
```

```
struct packet_header
{
    unsigned int iLen;
    unsigned int iProtocol;
};

struct pickup_item: public packet_header
{
    int iPlayerID;
    int iItemID;
};

pickup_item pk;
pk.iLen = sizeof(pickup_item);
pk.iProtocol = 1000;
pk.iPlayerID = 3020302;
pk.iItemID = 10203020;

write( m_hSocket, &pk, pk.iLen );
```

전송

```
struct packet_header
{
    unsigned int iLen;
    unsigned int iProtocol;
};

struct pickup_item: public packet_header
{
    int iPlayerID;
    int iItemID;
};

pickup_item pk;
read( m_hSocket, &pk, sizeof(pickup_item));
```

원격 프로시져 호출

RPC.NET은 IOCP기반으로 네트웍 이벤트를 처리하도록 구현되었으며 아래와 같은 간단한 코드 만으로도 다중 클라이언트를 처리할 수 있습니다.

```
Proactor* pProactor = ProactorFactory::Create( 0, 1, 2);
ProactorFactory::Start( pProactor->ID() );


Proactor::Property propCfg;
procCfg.sHost =  "localhost"
procCfg.iPort = 23344;
procCfg.pNetCallback = NULL
pProactor->Connect(  propCfg );

NetLinkPtr spLink = pProactor->Connect( 23344,  "localhost" );
```

연결

```
Proactor* pProactor = ProactorFactory::Create( 0, 1, 2);
ProactorFactory::Start( pProactor->ID() );


Proactor::Property propCfg;
procCfg.sHost =  "localhost"
procCfg.iPort = 23344;
procCfg.pNetCallback = NULL;
procCfg.iAcceptCount = 100
pProactor->Listen(  propCfg );
```

또한 RPC.NET 에서는 원격 프로시져 호출을 지원합니다. 서버에서는 원격에서 호출할 프로시져를 등록하고 클라이언트에서는 서버에 정의된 원격 프로시져를 추가 하는 것만으로 클라이언트에서 서버에게 원격 원격 프로시져를 실행하도록 요청할 수 있습니다.

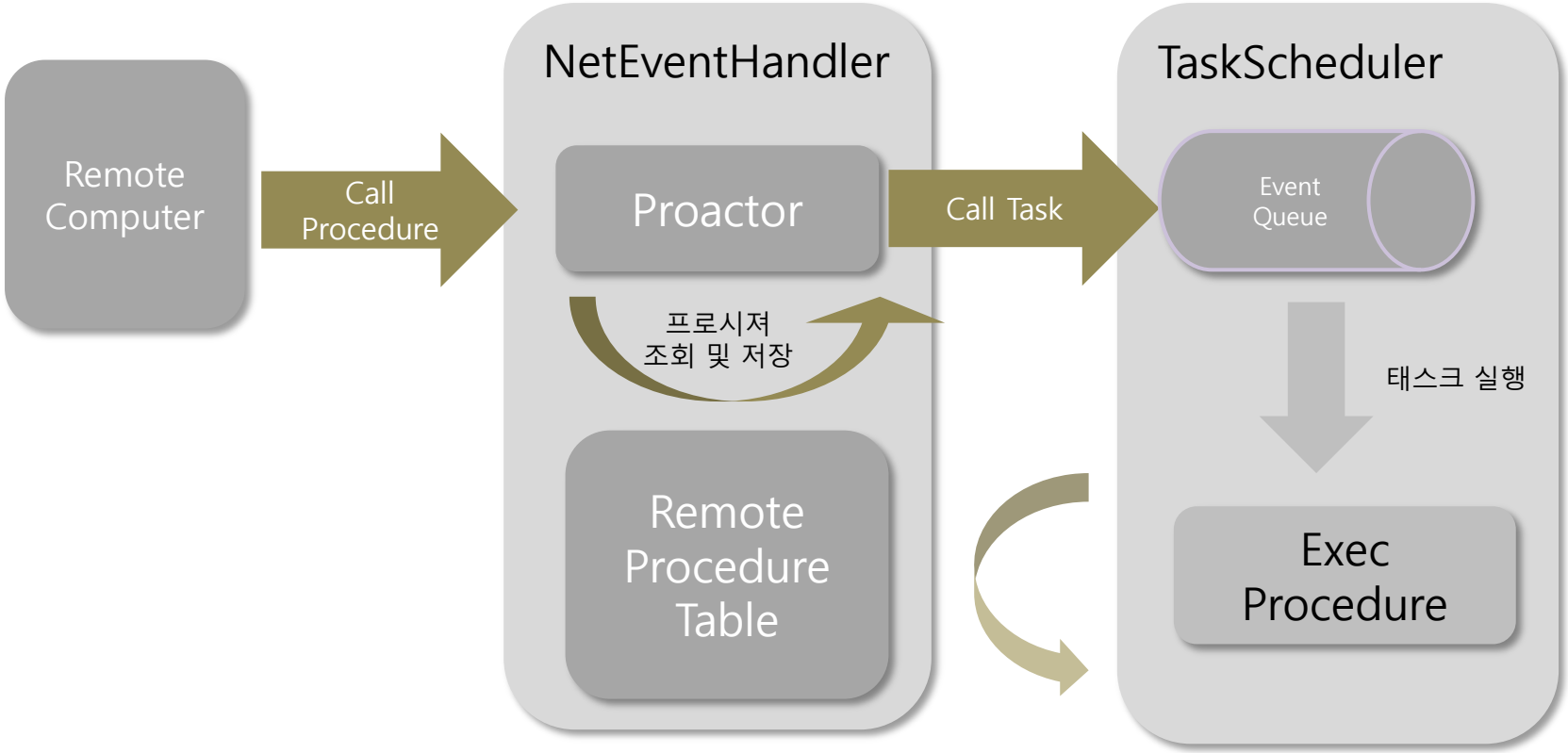
```
Class Server : public NetEventHandler
{
public:
    Server()
    {
        // 클라이언트에서 호출할 RPC 를 정의합니다.
        EXPORT_RPC_V2( 1,
                       2000,
                       this,
                       &Server::AddItem,
                       int,
                       int );
    }

    bool AddItem( IRef* pObj, int iPlayerID, int iItemID );
};
```

호출

```
Class Client : public NetEventHandler
{
public:
    Client()
    {
    }
    // 서버에 정의된 RPC 함수를 맴버로 정의합니다.
    IMPORT_RPC_V2( 2000, AddItem, int, int, 1024);
};

// 서버의 AddItem 을 원격 호출합니다.
rpc_AddItem( m_spLink, 3020302, 10203020 );
```

원격 프로시져 호출

RPC.NET은 호출할 원격 프로시져 ID 와 매개변수를 버퍼에 저장한 후 원격 프로시져를 호출하라는 명령을 만들어 네트웍을 통해서 원격호스트에게 전송합니다. 원격호스트의 NetEventHandler는 패킷을 분석하여 원격 프로시져와 매개변수를 꺼내와 작업이벤트에 등록한 후 태스크스케줄러에 전달합니다. 태스크스케줄러는 작업이벤트에 등록된 원격 프로시져를 실행합니다.
아래의 도식은 RPC.NET의 원격프로시져 호출과정을 도식으로 보여줍니다.



원격 프로시져 호출

RPC.NET 에서는 비동기적으로 발생하는 네트웍 이벤트 처리를 위해서 IOCP 기반으로 구현된 Proactor 클래스와 Proactor의 Wrapper 인 NetEventHandler 클래스을 제공합니다. 서버에서는 Listen 함수를 호출하여 클라이언트의 접속을 받아들일 준비를 하며 클라이언트에서는 Connect 함수를 호출하여 서버에 연결합니다. 또한 NetEventHandler 클래스는 원격 프로시져 호출 기능을 제공합니다. 아래는 RPC.NET의 네트웍 라이브러리를 구성하는 클래스들에 대한 설명입니다.

클래스	설명
NetLink	RPC.NET은 클라이언트와 서버간에TCP/IP 연결이 구축되거나 UDP 소켓 생성 또는 P2P 통신 설정이 완료되면 원격의 대상과 데이터를 송수신을 위한 네트웍 객체인 NetLink 를 생성하여 반환합니다. NetLink는 네트웍주소, 상태정보와 연결 대상에게 데이터를 송수신하기 위한 인터페이스들을 제공한다.
Proactor	IOCP기반의 Proactor 패턴으로 구현된 네트웍 이벤트 처리 클래스로 TCP/IP , UDP 소켓 생성 및 연결 구축, 데이터 송수신을 위한 인터페이스를 제공하며, 소켓에서 발생한 이벤트를 NetEventHandler에 전달하는 기능을 수행합니다.
NetEventHandler	NetEventHandler는 Proactor의 Wrapper 클래스로 Proactor에서 수신한 패킷를 분석하여 작업이벤트로 생성해서 태스크스케줄러에게 전달하는 기능을 수행한다. 만약 수신된 패킷이 원격 프로시져의 실행을 요청한 것이라면 원격프로시저 실행하라는 명령을 담은 작업이벤트를 태스크스케줄러에 전달하는 역할도 수행한다.

NetCallback	Proactor와 NetEventHandler는 소켓에서 발생한 클라이언트 접속, 서버 연결, 패킷 송수신, 종료와 같은 네트웍이벤트들을 처리하기 위해서 Listen(), Bind(), Connect() 함수 호출시 전달받은 NetCallback 객체의 네트웍이벤트 함수를 콜백한다.
	struct NetCallback
	{
	/** 클라가 연결되면 발생되는 네트웍이벤트*/
	virtual void OnNetAccepted(NetLinkPtr spLink) = 0;
	/** 원격호스트에 연결되면 발생되는 네트웍이벤트*/
	virtual void OnNetConnected(NetLinkPtr spLink) = 0;
	/** 패킷을 수신하면 발생되는 네트웍이벤트*/
	virtual void OnNetReceived(NetLinkPtr spLink, Buffer* pBuf) = 0;
	/** 연결이 종료되면 발생되는 네트웍이벤트*/
	virtual void OnNetClosed(NetLinkPtr spLink) = 0;
	};

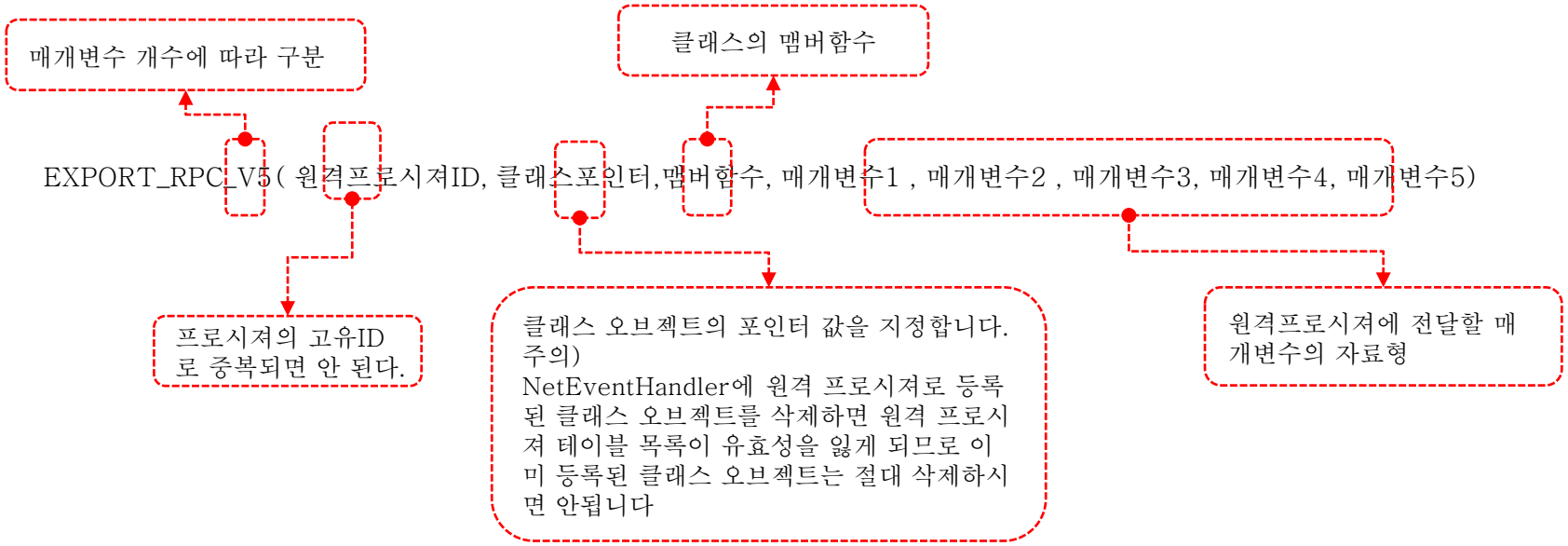
ProactorFactory	Proactor 객체 생성, 시작, 종료하는 전역 함수를 제공하는 네임스페이스

원격 프로시져 호출

NetEventHandler에 원격 프로시져를 등록하기 위해서는 EXPORT_RPC_ 로 시작하는 6 가지 매크로를 사용해야 합니다. 각각의 등록 매크로들은 원격 프로시져에게 몇 개의 매개변수를 전달할지에 따라 매크로명의 뒤에 V0 ~ V5 가 붙어 있을 뿐 기본 기능은 모두 동일합니다. 주의할 점은 NetEventHandler에 원격 프로시져로 등록된 클래스 오브젝트를 삭제하면 원격 프로시져 테이블 목록이 유효성을 잃게 되므로 이미 등록된 클래스 오브젝트는 절대 삭제하시면 안됩니다. 아래는 등록 매크로의 원형과 원격 프로시져로 등록할 함수의 원형을 보여줍니다.

- **EXPORT_RPC_V(원격프로시져ID, 클래스포인터, 맴버함수, 매개변수자료형들….)**
- **bool 함수명(IRef* , 매개변수들)**

매개변수	설명
원격프로시져 ID	원격 프로시져 테이블에 검색 키 값으로 사용되는 고유한 ID 입니다. IMPORT_RPC_ 매크로의 프로시져 ID 와 동일하게 매칭되게 정의한 후에 IMPORT_RPC_ 매크로에 정의한 함수이름을 호출하면 EXPORT_RPC_ 로 등록한 원격 프로시져가 호출됩니다.
클래스포인터	원격 프로시져로 등록할 맴버함수를 가지고 있는 클래스 오브젝트의 포인터 값입니다.
맴버함수포인터	원격 프로시져로 등록할 맴버함수의 포인터 값입니다.

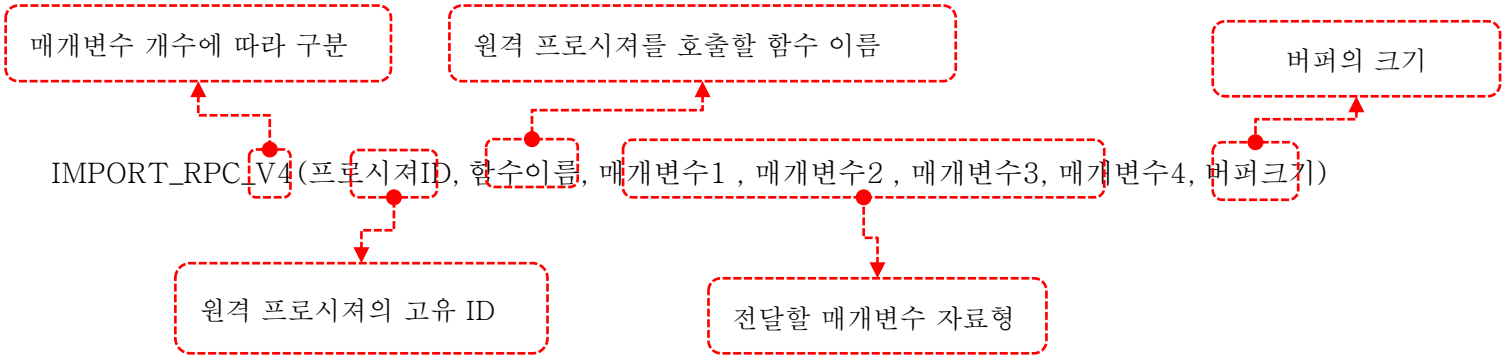


원격 프로시져 호출

NetEventHandler의 원격 프로시져를 호출 하기 위해서는 IMPORT_RPC_ 로 시작하는 6 가지 매크로를 사용하여 원격 프로시져에 대응하는 함수를 선언해야 합니다. IMPORT_RPC_ 매크로로 선언된 함수를 호출하면 네트웍을 통해서 NetEventHandler 에게 원격 프로시져를 실행하라는 작업이벤트가 전달됩니다. 아래는 선언 매크로의 원형을 보여줍니다. 한가지 주의할 점은 원격 프로시져를 호출할 때는 함수이름 앞에 rpc_ 를 붙여 호출해야 합니다.

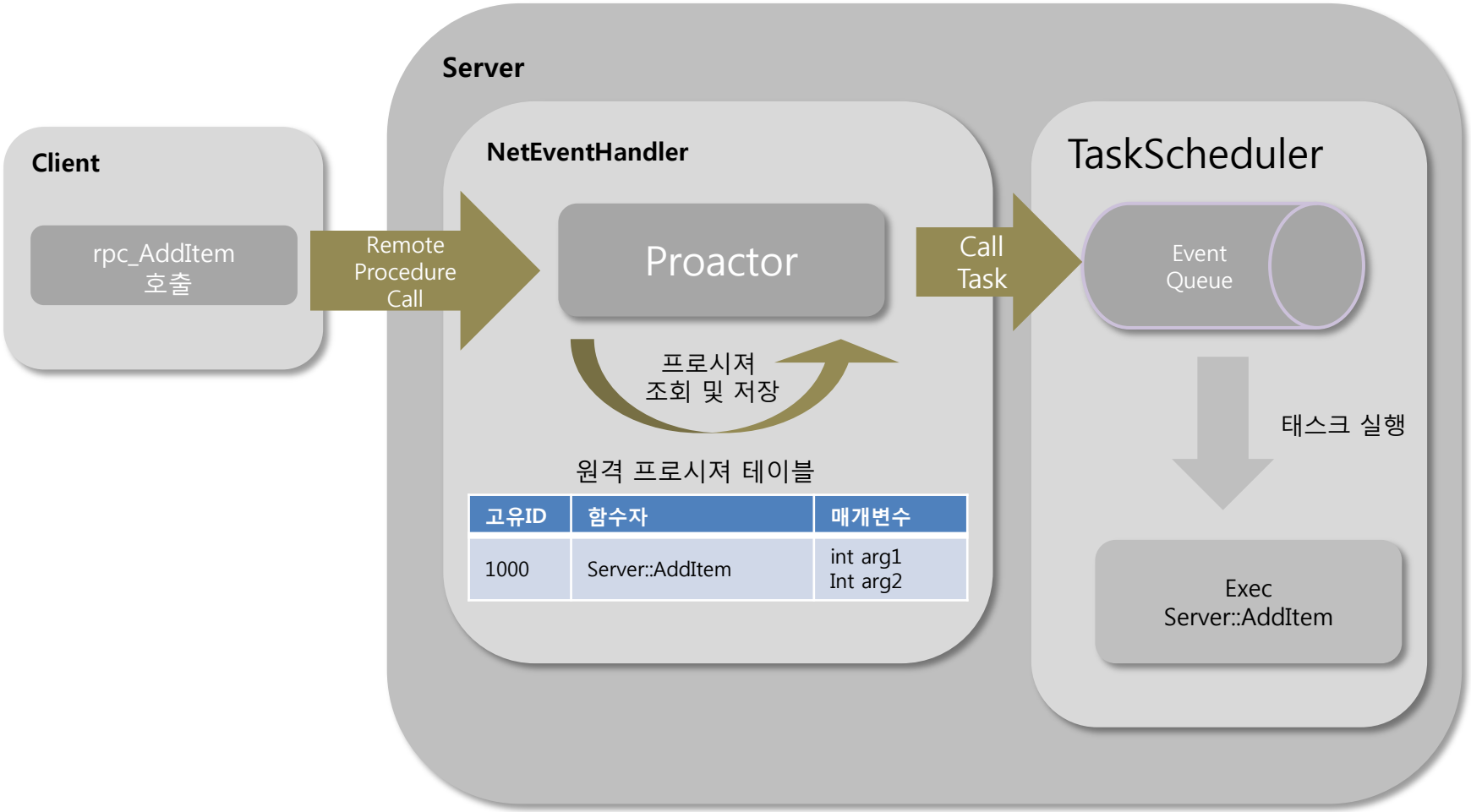
- **IMPORT_PROC_V(원격프로시져ID, 함수이름, 매개변수자료형들.., 버퍼크기)**
- **bool rpc_함수이름(NetLinkPtr, 매개변수들…)**

매개변수	설명
원격프로시져 ID	원격 프로시져 테이블에 검색 키 값으로 사용되는 고유한 ID 입니다. IMPORT_RPC_ 매크로의 프로시져 ID 와 동일하게 매칭되게 정의한 후에 IMPORT_RPC_ 매크로에 정의한 함수이름을 호출하면 EXPORT_RPC_ 로 등록한 원격 프로시져가 호출됩니다.
함수이름	원격 프로시져를 호출할 함수이름으로 IMPORT_RPC_V 매크로에 정의한 함수이름 앞에 rpc_ 붙여서 호출합니다. 예를 들어 아래와 같이 함수이름 func 으로 정의하면 호출할 때는 rpc_func 으로 호출해야 한다. IMPORT_RPC_V1(1000, func, int, 1024); rpc_func(spLink, 32);
버퍼크기	원격 프로시져를 호출할 때 생성되는 버퍼의 크기로 프로시져 ID 및 매개변수가 저장될 공간의 크기를 정의한다.



원격 프로시져 호출

지금까지 설명한 내용들의 이해를 돕기 위해서 간단한 예제를 하나 만들어 보겠습니다. 아래의 예제는 서버에서 원격 프로시져로 ItemManager::AddItem 를 등록하고 클라이언트에서 서버의 ItemManager::AddItem 을 실행하도록 요청하는 방법을 보여줍니다.



원격 프로시져 호출

서버에서 원격 프로시져로 ServerHandler::AddItem 맴버함수를 등록합니다. 이때 원격 프로시져의 고유ID 로 2000번 지정하였고 원격 프로시져에 전달할 매개변수의 자료형으로 int, int 를 지정했습니다.

NetEventHandler

원격 프로시져 테이블

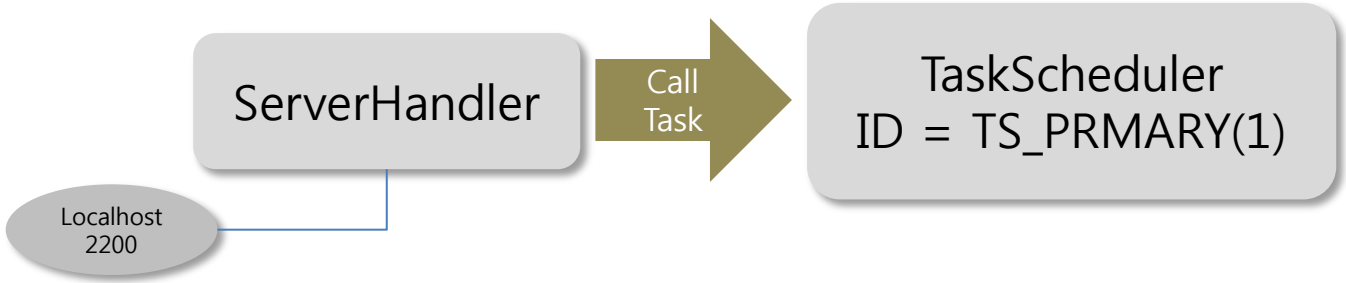
고유ID	함수자	매개변수
2000	Server:Handler:AddItem	int arg1 Int arg2

```
class ServerHandler : public NetEventHandler
{
public:
    ServerHandler()
    {
        EXPORT_RPC_V2(  2000, this, &Server::AddItem,  int, int  );
    }
    virtual ~ServerHandler()  {}

    // 원격 프로시져로 등록할 함수는 반드시 bool 함수명( IRef*, 매개변수.. ) 형태
    bool AddItem( IRef* pObj, int iPlayerID, int iItemID  )
    {
        printf( __FUNCTION__": Thread %d\n", ::GetCurrentThreadId() );
        return true;
    }
};
```

EXPORT_RPC_V2 매크로를 사용하여 ServerHandler::AddItem 함수를 고유ID 2000 번으로 원격 프로시져 테이블에 등록합니다.

원격 프로시져 호출

이제 서버의 원격 프로시져 호출할 준비를 마쳤으니 서버에서 클라이언트의 연결을 받아 들이도록 구현합니다. 그리고 1번 태스크 스케줄러에 ServerHandler::AddItem 맴버함수를 원격 프로시져로 등록하였으므로 태스크스케줄러를 1번으로 생성해야 합니다.




```
#include <Proactor.h>
const unsigned int TS_PRIMARY = 1

int _tmain(int argc, _TCHAR* argv[])
{
   TaskScheduler* ts = CreateTaskScheduler(TS_PRIMARY, 1, 1);
   StartTaskScheduler(TS_PRIMARY );

   ServerHandler handler(TS_PRIMARY);
   handler.Start(1,1,2);

   handler.Listen( 2200,  "localhost" );
    getchar();

   handler.Stop();
   StopTaskScheduler();

    return 0;
}
```

태스크스케줄러를 생성해서 실행합니다.

네트워크 이벤트 핸들러를 생성해서 실행합니다.

클라이언트가 접속할 수 있도록 로컬에 포트를 개방합니다.

네트워크 이벤트 핸들러 및 태스크스케줄러를 종료합니다.

원격 프로시져 호출

클라이언트가 서버에 연결되면 서버와 연결된 NetLink를 m_spLink 에 저장한후 1초에 한번씩 OnTimer가 호출되도록 합니다. OnTimer 함수에서는 rpc_AddItem 을 호출하여 서버에서 원격 프로시져가 실행되도록 요청합니다. 아래는 서버에 정의한 원격프로시져를 호출하는 클라이언트를 구현한 것입니다.

```
class ClientHandler : public NetEventHandler
{
    NetLinkPtr     m_spLink;
public:
    ClientHandler()
    {
        SetConnectedFunc( this, &ClientHandler::OnConnected );
    }
    virtual ~ ClientHandler () {}

    IMPORT_RPC_V2( 1000, AddItem, int, int );

    bool Connected( NetLinkPtr spLink )
    {
        m_spLink = spLink;

        CreateTimer(1, this, &ClientHandler::OnTimer, 1000, NULL );
        return true;
    }

    bool OnTimer( void * pUsrData )
    {
        rpc_AddItem( m_spLink, 100, 102 );
        return true;
    }
};
```

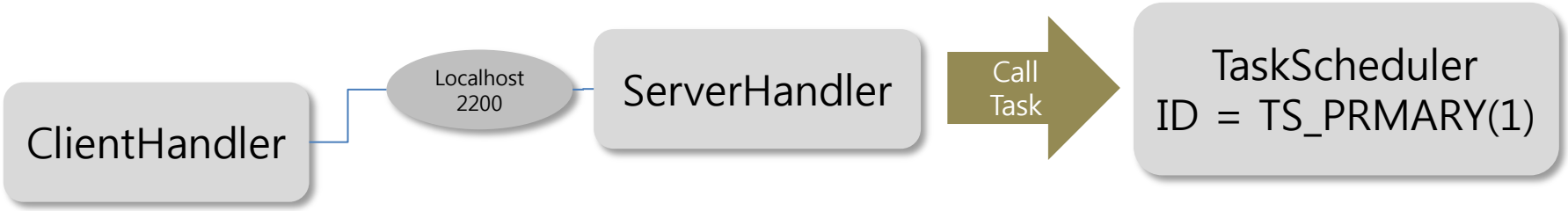
IMPORT_RPC_V2 매크로를 사용하여 Server::AddItem 함수를 호출할 함수를 맴버함수로 추가합니다.

1 초에 한번씩 OnTimer 를 호출하도록 타이머를 생성합니다.

Server::AddItem 함수의 호출을 요청합니다.

원격 프로시져 호출

마지막으로 클라이언트에서 서버로 연결하는 코드입니다.




```
#include <Proactor.h>
const unsigned int TS_PRIMARY = 1

int _tmain(int argc, _TCHAR* argv[])
{
   TaskScheduler* ts = CreateTaskScheduler(TS_PRIMARY, 1, 1);
   StartTaskScheduler(TS_PRIMARY );

   ClientHandler handler(TS_PRIMARY);
   handler.Start(1,1,2);

   handler.Connect ( 2200, "localhost" );
   getchar();

   handler.Stop();
   StopTaskScheduler();

   return 0;
}
```

태스크스케줄러를 생성해서 실행합니다.

네트워크 이벤트 핸들러를 생성해서 실행합니다.

클라이언트가 접속할 수 있도록 로컬에 포트를 개방합니다.

네트워크 이벤트 핸들러 및 태스크스케줄러를 종료합니다.

원격 프로시져 호출

NetEventHandler 의 API 목록은 아래와 같습니다.

함수명	함수 원형	설명
SetAcceptedFunc	template<typename O, typename F> inline void SetAcceptedFunc(const O& o, F f) Ex) SetAcceptedFunc(this, &ServerHandler::OnAccepted);	클라이언트가 연결되면 호출할 콜백함수 설정한다 -. o 클래스 객체의 주소값 -. f 클래스 객체의 맴버함수의 주소값
SetClosedFunc	template<typename O, typename F> inline void SetClostedFunc(const O& o, F f) Ex) SetClosedFunc(this, &ServerHandler::OnClosed);	연결이 종료되면 호출할 콜백함수 설정한다 -. o 클래스 객체의 주소값 -. f 클래스 객체의 맴버함수의 주소값
SetConnectedFunc	template<typename O, typename F> inline void SetConnectedFunc(const O& o, F f) Ex) SetConnectedFunc(this, &ServerHandler::OnReceived);	서버에 연결되면 호출할 콜백함수 설정한다 -. o 클래스 객체의 주소값 -. f 클래스 객체의 맴버함수의 주소값
SetReceivedFunc	template<typename O, typename F> inline void SetReceivedFunc(const O& o, F f) Ex) SetReceivedFunc(this, &ServerHandler::OnReceived);	네트웍에서 패킷을 수신하면 호출할 콜백함수를 설정한다. 만약 콜백할 함수가 설정되면 네트웍에서 발생한 모든 패킷을 우선 처리하므로 네트웍이벤트함수와 원격프로시져 호출이 수행되지 않는다. -. o 클래스 객체의 주소값 -. f 클래스 객체의 맴버함수의 주소값
SetNetFunc	template<typename O, typename F> inline void SetReceivedFunc(unsigned int iProtocol, const O& o, F f) Ex) SetNetFunc(10000,this, &ServerHandler::OnAddItem);	패킷ID로 식별되는 네트웍이벤트함수를 설정한다. 만약 네트웍이벤트함수의 패킷ID와 원격프로시져의ID가 중복될 경우 네트웍이벤트함수가 호출 우선순위를 갖기 때문에 원격프로시져는호출되지 않는다. -. iProtocol 패킷ID -. o 클래스 객체의 주소값 -. f 클래스 객체의 맴버함수의 주소값

Start	bool Start(unsigned int iPriority, unsigned short iBindCPU, unsigned short iThreadCnt);
	Ex)
	NetEventHandler evh(1);
	evh.Start(0, // 종료 우선순위
	1, // 네트웍 쓰레드를 실행될 CPU 번호
	2); // 네트웍 쓰레드 개수
		Proactor를 생성하여 네트웍이벤트 감시를 시작한다.
		-. iPriority 네트웍를 종료시키는 우선순서
		-. iBindCPU 네트웍쓰레드가 실행될 CPU
		-. iThreadCnt 네트웍쓰레드 개수

원격 프로시져 호출

NetEventHandler 의 API 목록은 아래와 같습니다.

함수명	함수 원형	설명

Stop	void Stop(void);
	Ex)
	NetEventHandler evh(1);
	evh.Stop();
		NetEventHandler가 관리하고 있는 모든 소켓을 종료한후 Proactor를 종료한다.

Listen	bool Listen(unsigned short iPort, const std::string& sHost = "localhost"); Ex) Bool bRet = Listen(25533, “localhost”);	TCP/IP 소캣을 생성한후 클라이언트와 연결을 구축할 준비한다. -. iPort 클라이언트의 접속을 받아들일 포트번호 -. sHost 클라이언트의 접속을 받아들일 주소값
Connect	NetLinkPtr Connect(unsigned short iPort, const std::string& sHost = "localhost"); Ex) NetLinkPtr spLink = Connect(25533, “localhost”);	TCP/IP 소캣을 생성한후 서버와 연결한다. -. iPort 서버의 포트번호 -. sHost 서버의 주소값
Bind	NetLinkPtr Bind(unsigned short iPort, const std::string& sHost = "localhost"); Ex) NetLinkPtr spLink = Bind(25533, “localhost”);	UDP 소캣을 생성한다. -. iPort 서버의 포트번호 -. sHost 서버의 주소값

원격 프로시져 호출

EXPORT_RPC 매크로의 정의는 아래와 같습니다.

매크로	선언방법
EXPORT_RPC_V0	EXPORT_RPC_V0(1000, this, &ItemManager::AddItem);
EXPORT_RPC_V1	EXPORT_RPC_V1(1001, this, &ItemManager::AddItem, int);
EXPORT_RPC_V2	EXPORT_RPC_V2(1002, this, &ItemManager::AddItem, int, int);
EXPORT_RPC_V3	EXPORT_RPC_V3(1003, this, &ItemManager::AddItem, int, int, const std::string&);
EXPORT_RPC_V4	EXPORT_RPC_V4(1004, this, &ItemManager::AddItem, int, int, int, float);
EXPORT_RPC_V5	EXPORT_RPC_V5(1005, this, &ItemManager::AddItem, int, int, double, int, float);

IMPORT_RPC 매크로의 정의는 아래와 같습니다.

매크로	선언방법
IMPORT_RPC_V0	IMPORT_RPC_V0(1000, AddItem);
IMPORT_RPC_V1	IMPORT_RPC_V1(1001, AddItem, int, 1024);
IMPORT_RPC_V2	IMPORT_RPC_V2(1002, AddItem, int, int, 1024);
IMPORT_RPC_V3	IMPORT_RPC_V3(1003, AddItem, int, int, const std::string&, 1024);
IMPORT_RPC_V4	IMPORT_RPC_V4(1004, AddItem, int, int, int, float1024);
IMPORT_RPC_V5	IMPORT_RPC_V5(1005, AddItem, int, int, double, int, float, 1024);

원격 프로시져 호출

원격 프로시져로 호출할 맴버함수들의 매개변수 자료형은 아래 목록과 같습니다.

구분	자료형	선언방법
문자열	std::string / const std::string&, mem::string / const mem::string&	EXPORT_RPC_V1(1, 1000, this, &TestObj2::Func2, const std::string&);
스칼라	char / unsigned char, short / unsigned short, int / unsigned int, long / unsigned long, __int64 / unsigned __int64, float, double	EXPORT_RPC_V1(1, 1001, this, &TestObj2::Func3, int);

구조체	Buffer 의 전역 연산자 << 와 >> 를 구현하여 구조체 전달할 수 있습니다.
	struct custom
	{
	int a1;
	std::string s1;
	};
	
	저장 연산자 구현
	Buffer& operator<<(Buffer& pk, const custom& r)
	{
	pk << r.a1 << r.s1;
	return pk;
	}
	
	Buffer& operator>>(Buffer& pk, const custom& r)
	{
	pk >> r.a1 >> r.s1;
	return pk;
	}
		EXPORT_RPC_V1(1, 1003, this, &TestObj2::Func3, const custom&);

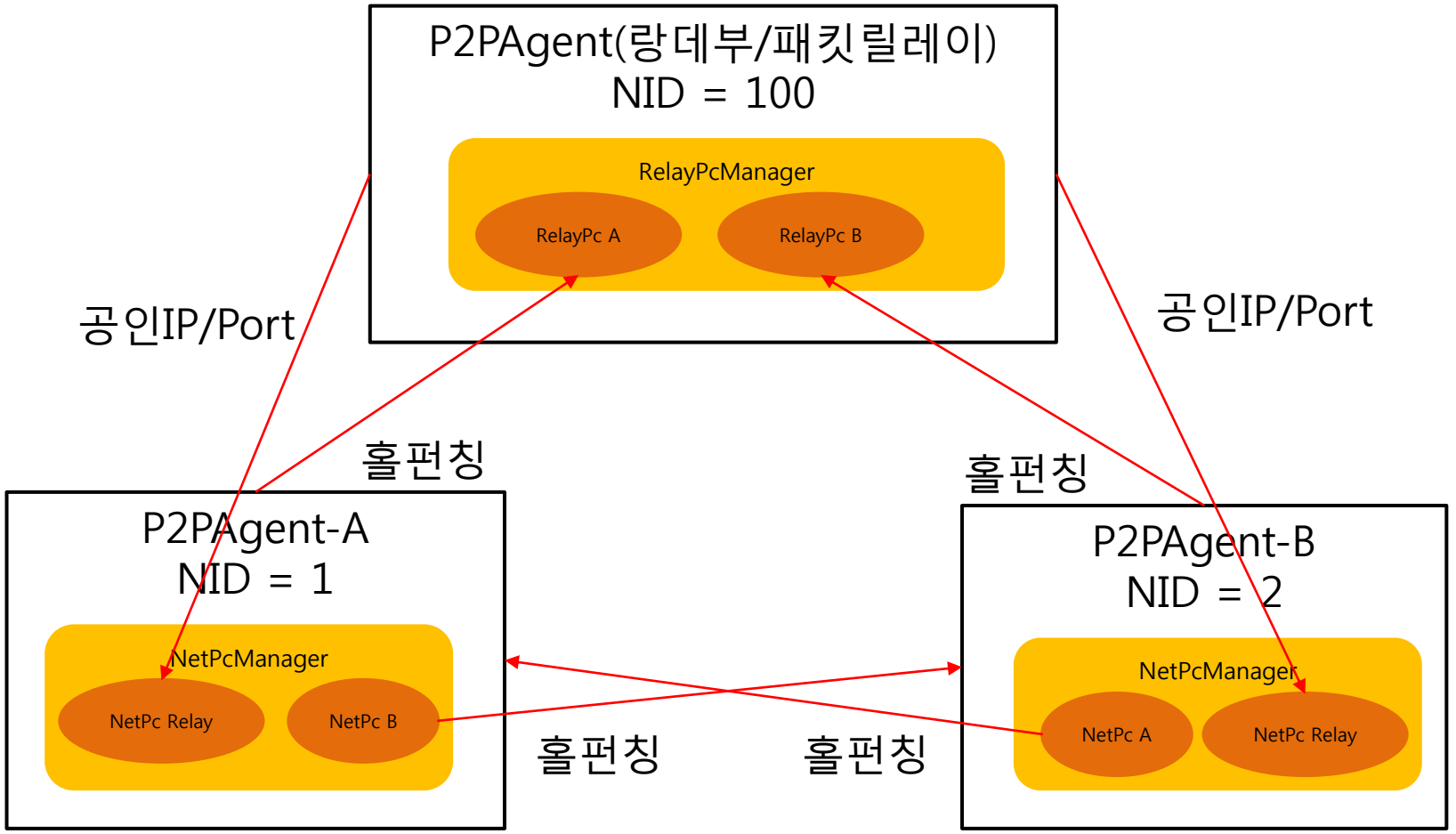
P2P 네트웍

RPC.NET은 STUN을 통해 NAT의 종류를 판별하지 않습니다. 랑데부 서버에 로그인한 원격PC간에 네트웍 정보를 공유하고 서로간에 홀펀칭을 수행하며, 홀펀칭이 안된 원격PC에는 릴레이서버로 통신을 합니다. Restrict – Full, Restrict – Restrict, Full – Restrict, Symetric – Full 간에는 모두 홀펀칭이 정상적으로 되지만 Restrict – Symetric , Symetric – Symetric 은 홀펀칭이 실패되므로 릴레이서버로 통신을 합니다.

클래스명	설명
Network_IF	원격PC의 사용자ID/IP,Port 정보를 관리한다.
NetLink	원격PC를 가리키는 네트웍 객체를 나타낸다.
Pc	NetPc 및 RelayPc 의 상위 클래스로 홀펀칭상태 관리, P2P 패킷 버퍼링 및 전송할 패킷의 일련번호 관리를 한다.
NetPc	랑데뷰서버로 부터 수신한 control 패킷 처리한다.
RelayPc	랑데뷰서버에서 원격PC의 Join/Leave ack 패킷을 처리 및 패킷 릴레이를 수행한다.
P2PAgent	NetLinkManager 상위 추상 클래스로 RPC.NET에서 P2P 통신을 위한 인터페이스를 제공한다.
NetLinkManager	NetPcManager/RelayPcManager의 상위 클래스로 원격 PC(NetPc/RelayPc) 관리자이다. 원격 PC에서 수신한 패킷을 NetPc/RelayPc 로 전달하며 홀펀칭 및 연결유지를 수행한다.
NetPcManager	NetPc 를 생성하여 NetLinkManager 로 전달한다.
RelayPcManager	RelayPc 를 생성하여 NetLinkManager 로 전달하며 원격PC의 사용자ID,공인IP/Port 정보와 접속 종료 상태 감지하며 및 원격 PC 에서 수신한 패킷을 릴레이 하는 기능을 수행한다.
P2PAgentFactory	P2P통신을 위한 P2PAgent 생성
RendezvousAgentFactory	랑데뷰/패킷릴레이 서버 생성하고 랑데뷰/패킷릴레이 서버 연결하는 인터페이스를 제공한다.

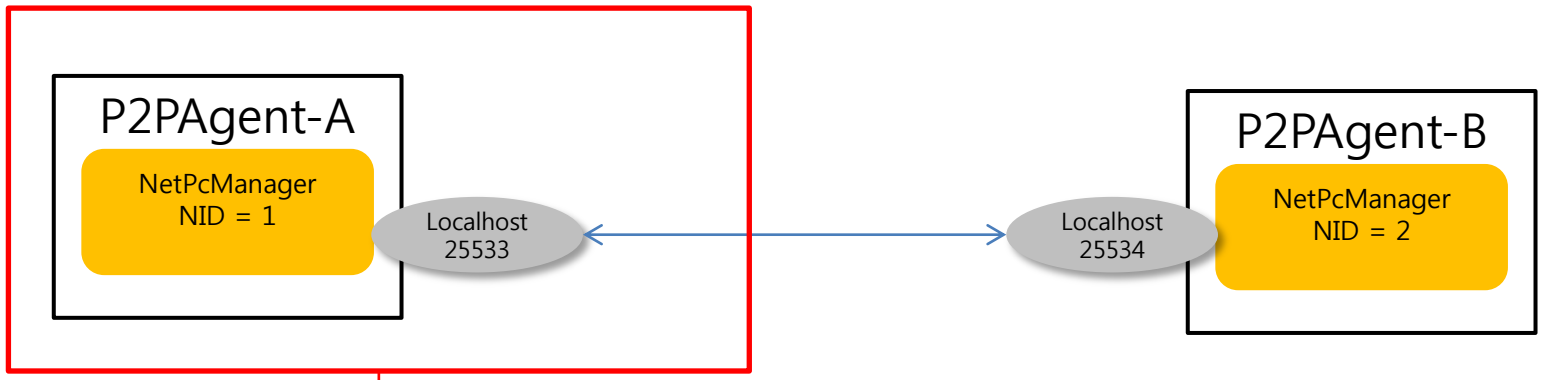
P2P 네트웍

P2PAgent는 랑데뷰서버로 자신의 공인 IP, Port 정보를 알아내기 위해 패킷을 전송하고 랑데뷰서버는 P2PAgent의 사용자ID, 네트웍 정보를 등록한 후 현재 등록된 모든 P2PAgent에게 네트웍 정보 및 사용자ID를 브로드캐스팅 한다. P2PAgent는 랑데뷰서버에서 수신된 사용자ID,네트웍 정보를 등록한 후 원격 P2PAgent들과의 홀펀칭을 합니다. 홀펀칭에 성공한 원격 P2PAgent와는 P2P 통신을 하며 그렇지 않은 원격 P2PAgent와는 릴레이서버를 경유하여 패킷릴레이를 합니다.



P2P 네트웍

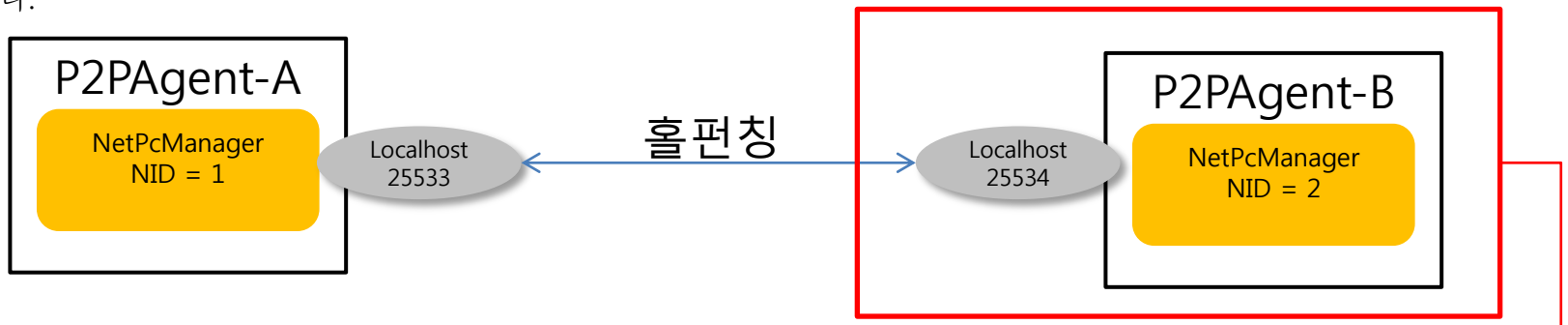
상대방의 네트웍 고유 ID와 공인 IP,Port 정보를 모두 알고 있는 경우 홀펀칭을 수행하기 위해서는 먼저 P2PAgent를 생성해야 합니다. 최초 P2PAgent 가 생성되면 아래 도식의 왼쪽과 같이 네트웍고유ID(NID) = 1 그리고 네트웍주소값은 IP = localhost 이고 Port = 25533 입니다.




```
// 네트웍 고유ID는 1, 네트웍 주소값은 IP = 127.0.0.1 Port = 25533 으로 P2PAgent를 생성합니다.
Network_IF nif(1, "127.0.0.1", 25533);
pAgent = P2PAgentFactory::Create( nif, &Hdr );
if ( !pAgent ) return 0;

printf( "고유ID: %u, 네트웍정보: %s₩n", pAgent->Self().m_iNID, NetLink::tostr(&pAgent->Self()).c_str() );

// 홀펀칭 및 P2P 패킷의 송수신 수행합니다.
while (1)
{
    pAgent->Process( 1 );
}
```

P2P 네트웍

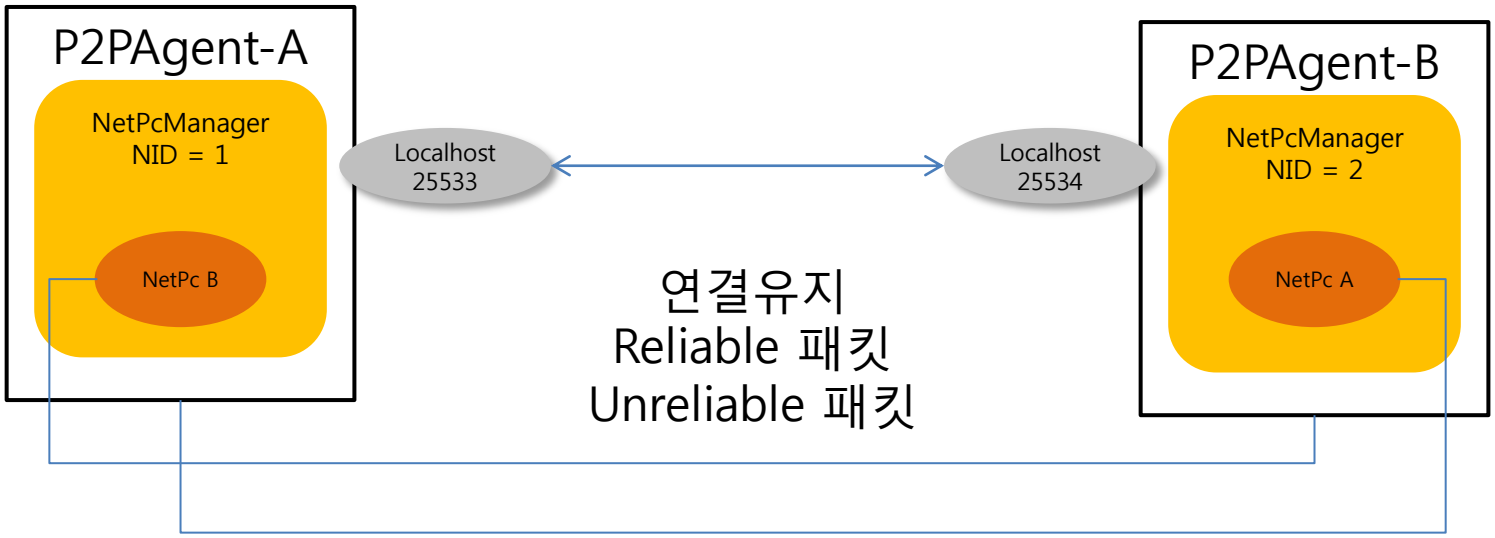
상대방과 홀펀칭을 수행하기 위해서 자신의 네트웍 정보를 설정하여 P2PAgent을 생성한 후 원격 pc의 네트웍 정보를 지정합니다.




```
// 네트웍 고유ID는 2, 네트웍 주소값은 IP = 127.0.0.1 Port = 25534 으로 P2PAgent를 생성합니다.
Network_IF nif(2, "127.0.0.1", 25534);
pAgent = P2PAgentFactory::Create( nif, &Hdr );
if ( !pAgent ) return 0;

printf( "고유ID: %u, 네트웍정보: %s₩n", pAgent->Self().m_iNID, NetLink::tostr(&pAgent->Self()).c_str() );

// 홀펀칭을 수행할 원격 pc 의 네트웍 정보를 설정합니다.  네트웍고유ID = 1, 네트웍주소값 IP = 127.0.0.1, Port = 25533
Network_IF peer_nif(1, "127.0.0.1", 25533);
NetLink* pLink = pAgent->Connect( peer_nif );

unsigned long iSendTick = 0;
while (1)
{
    if ( pLink && pLink->NetST() == eLINK_ST && iSendTick <= timeGetTime() )
    {
        char szBuf[1024];
        int iLen = sprintf_s(szBuf, 1024, "this is test" );
        pLink->Push( szBuf, iLen+1, true );
        iSendTick = timeGetTime() + 1000;

    }
    pAgent->Process( 1 );
}
```

P2P 네트웍

원격의 P2PAgent간에 홀펀칭이 완료되면 P2PAgent에 상대방의 P2PAgent를 가리키는 NetPc가 생성됩니다. NetPc에는 원격의 P2PAgent 네트웍 정보와 연결상태 및 reliable/unreliable 패킷 일련번호 정보를 가지고 있으며 NetPcManager가 이 정보를 참조하여 원격 P2PAgent와 연결을 유지하고 P2P 패킷을 송수신하게 됩니다.



P2P 네트웍

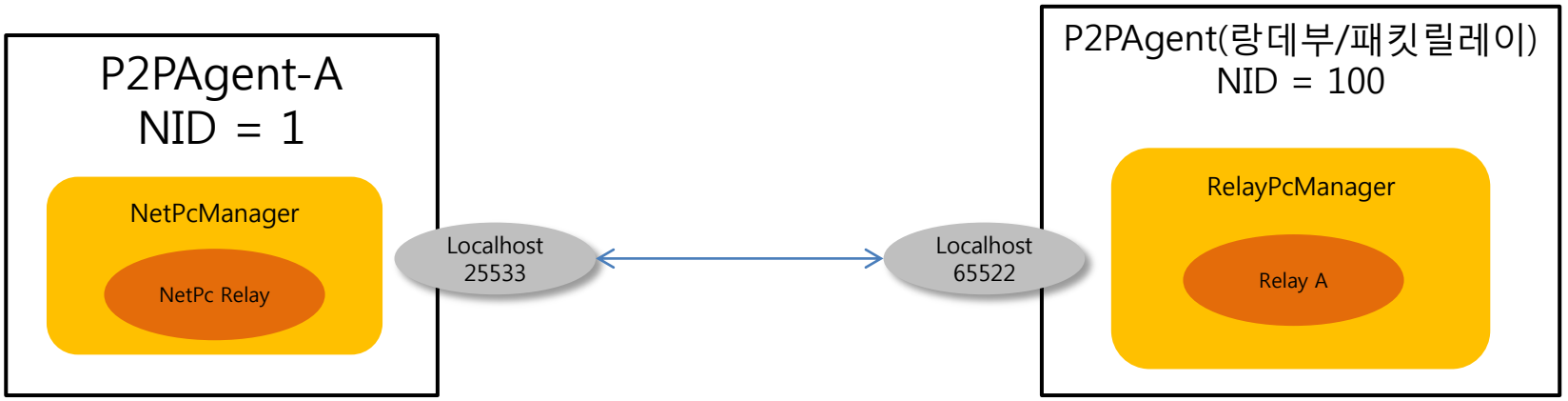
앞장에서는 원격 P2PAgent의 네트웍고유ID 및 공인 IP,Port를 알고 있는 경우에 적용할 수 있는 방법이면 원격 P2PAgent의 네트웍 정보를 모르는 경우 랑데부 서버를 통해서 네트웍 정보를 얻어와야 합니다. RPC.NET의 랑데부/패킷릴레이 서버는 연결된 원격 P2PAgent들에게 네트웍 정보를 브로드캐스팅하여 홀펀칭에 필요한 정보를 공유하도록 합니다. 또한 홀펀칭이 실패한 원격 P2PAgent간에는 패킷을 릴레이 해주는 역할도 수행합니다.


```
// 네트웍고유ID = 100, IP = 127.0.0.1,Port: = 65522로 랑데뷰/패킷릴레이서버를 생성한다.
Network_IF nif(100, "127.0.0.1", 65522);
pAgent = RendezvousAgentFactory::Create( nif );
if ( !pAgent ) return 0;

printf( "랑데뷰/패킷릴레이고유ID: %u, 네트웍정보: %s₩n", pAgent->Self().m_iNID, NetLink::tostr(&pAgent->Self()).c_str() );

While( 1 )
{
    pAgent->Process(1);
}
```

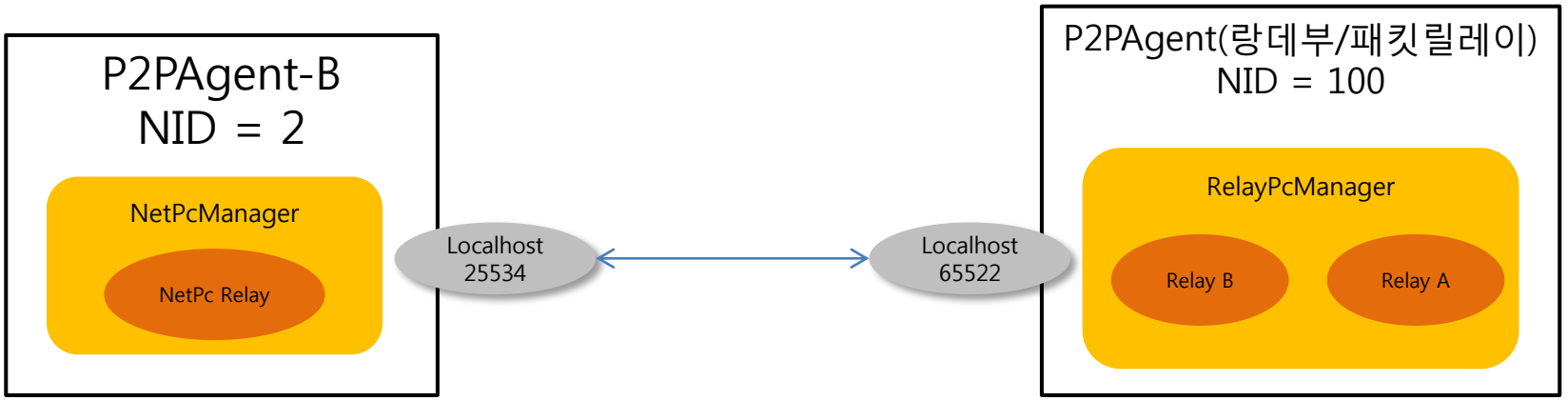
P2P 네트웍

랑데부/패킷릴레이 서버에 연결된 자신의 공인IP/Port 및 원격 P2PAgent의 네트웍고유ID 및 공인 IP/Port 정보를 얻어와 홀펀칭을 수행합니다.

P2PAgent –A가 랑데부 서버에 연결되면 랑데뷰서버에는 RelayPc A 가 생성되어 P2PAgent–A 를 가리키게 됩니다.

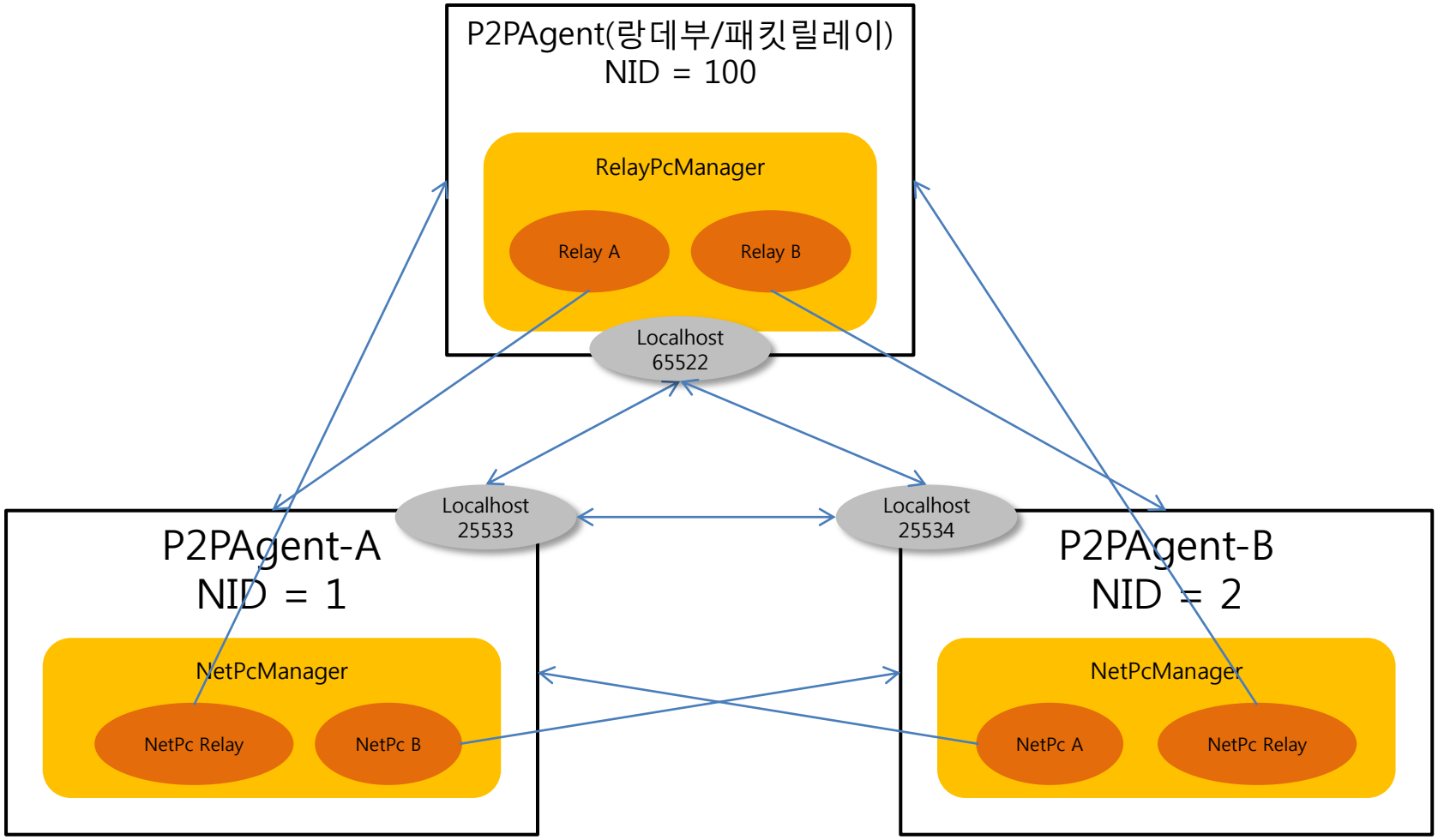


P2PAgent –B 가 랑데부 서버에 연결되면 랑데부서버에는 RelayPc B 가 생성되어 P2PAgent–B를 가리키게 됩니다. 그리고 랑데부 서버에 연결된 모든 P2PAgent 네트웍 정보를 브로드 캐스팅 합니다.



P2P 네트웍

랑데부서버로 부터 원격 P2PAgent의 정보를 수신한 P2PAgent들은 서로 홀펀칭을 수행합니다. 아래는 홀펀칭이 완료된 후의 P2PAgent 를 보여줍니다.



P2P 네트웍

랑데부서버로 연결하는 코드의 구현은 아래와 같습니다.

```
// 자신의 네트웍정보를 설정한다.
Network_IF nif(atoi(argv[1]), "127.0.0.1", atoi(argv[2]));
// 랑데뷰/패킷릴레이서버의 네트웍정보를 설정한다.
Network_IF rendezvous_nif(100, "127.0.0.1", 65522 );
// 랑데뷰/패킷릴레이서버에 접속한다.
pAgent = RendezvousAgentFactory::Join( nif, &Hdr, rendezvous_nif );
if ( !pAgent ) return 0;

printf( "고유ID: %u, 네트웍정보: %s₩n", pAgent->Self().m_iNID, NetLink::tostr(&pAgent->Self()).c_str() );

while (1)
{
    // 패킷 처리및 연결유지를 수행한다.
    pAgent->Process(1);
}
```

무잠금 메모리풀

외부 단편화는 할당된 메모리 블록 사이에 빈 공간이 있을 때 생겨난다. 만약 애플리케이션이 연속된 세 블럭의 메모리를 할당한 후 가운데를 제가할 때에 외부 단편화가 일어난다. 가운데 블록을 다시 사용할 수도 있지만 모든 자유 메모리 만큼이나 큰 블록을 할당하는 것은 불가능하다.

정리를 하자면 3이라는 공간에 10을 넣는다 할때 마지막 1이 남는데 이것을 내부 단편화라 하고, 3씩 메모리를 3개 할당한 후 2번째 것을 해제하고 4를 넣는다면 해제한 2번째 공간은 비워두고 새로운 공간에 4를 할당하는 것을 외부 단편화라 볼 수 있다.

메모리 단편화를 해결하기 위해서 메모리 풀링 과 같은 방식을 사용한다. 메모리 풀링 이란 동적인 메모리 할당과 해제를 줄이는 방법으로 큰 메모리를 할당해 놓고 필요할 때마다 꺼내어 사용하고 다 사용한 메모리는 다시 넣어놓는 과정을 말한다.

TLS (Thread Local Storage)
한 프로세스에 속한 쓰레드들은 주소 공간과 핸들 테이블을 공유하기 때문에 한 쓰레드가 전역 변수를 변경하면 다른 쓰레드도 이 변경의 영향을 받는다. 쓰레드 지역 지역 장소인 TLS는 스레드가 만들어질 때마다 OS가 각 스레드별로 LPVOID의 배열을 할당하고 모든 배열 요소를 NULL로 초기화 한다. 각 쓰레드에서 TLS에 메모리를 할당하면 OS는 비어 있는 배열에 할당한 메모리를 저장한 후 배열의 인덱스를 돌려준다. 쓰레드 에서 할당된 TLS 에 접근하기 위해서는 할당시 전달받은 인덱스를 사용하여 메모리에 접근한다.

TLS를 사용하는 좀 더 편리한 방법은 마이크로소프트가 확장한 지정자를 사용하는 것이다.
전역, 정적 변수 선언문 앞에 __declspec(thread) 키워드를 붙이면 이 변수는 쓰레드 별로 별도의 기억 장소를 할당 받는다. 지역 변수나 함수, 멤버 함수를 가진 클래스에는 이 지정자를 붙일 수 없다. 이 키워드로 선언된 변수는 .tls라는 별도의 섹션에 저장된다. 단, 이 방법은 DLL을 명시적으로 연결할때는 적용되지 않는다. 명시적으로 연결되는 DLL은 TlsAlloc류의 함수를 사용하는 수밖에 없다.

```
DWORD WINAPI TlsAlloc(void);
LPVOID WINAPI TlsGetValue(_In_  DWORD dwTlsIndex);
BOOL WINAPI TlsFree(_In_  DWORD dwTlsIndex)
```

무잠금 메모리풀

무잠금 메모리풀 (Lock Free Memory Pool)

메모리 풀(memory pool)란 고정된 크기의 블록을 할당하여 malloc이나 C++의 new 연산자와 유사한 메모리 동적 할당을 가능하게 해준다. malloc이나 new 연산자 같은 기능들은 다양한 블록사이즈 때문에 메모리단편화를 유발하고, 단편화된 메모리들로 인해 Application의 성능이 저하된다.
무잠금 메모리 풀은 멀티쓰레드 환경하에서 시스템의 제한된 메모리를 효율적으로 사용하기 위해TLS (Thread Local Storage) 내에 무잠금 메모리 풀을 생성한다. 따라서 쓰레드들은 메모리 할당 및 반납시에 쓰레드간의 경합(Lock/Unlock)을 제거하여 Application의 성능이 향상되는 효과를 볼 수 있다.

메모리 단편화(Memory Fragmentation)

단편화란 어떠한 공간이나 자료가 여러 조각으로 나뉜다는 것을 의미한다.그리고 단편화된 메모리란 "사용할 수 없는 자유메모리"로 설명될 수 있다.
자유 메모리는 메모리 할당자가 해당 메모리를 적재할 수 없는 상태로 만들 때 생겨나며, 이들이 불연속적인 작은 부분에 따로 흩어져 있기 때문에 메모리 단편화 문제가 발생한다. 할당 방법에 따라 메모리 단편화가 문제되기도 하고 그렇지 않을 수도 있으므로, 메모리 할당자는 사용 가능한 리소스의 가용성을 높이는데 중요한 역할을 한다.

컴파일러 및 링커가 메모리 할당기능을 수행할 때에는 메모리 단편화가 일어나지 않는다. 이는 컴파일러가 데이터의 수명을 알고 있기 때문이다. 데이터의 수명을 알고 있으면 스택 영역에 배치할 수 있는 이점이 있다. 이런 이유로 메모리 할당자는 메모리 단편화 없이 효율적으로 작업할 수 있는 것이다. 일반적으로 런타임시 메모리 할당은 스택될 수 없다. 그리고 메모리 할당은 시간과는 무관한 측면이 있어서 단편화 문제의 해결을 어렵게 한다. 메모리 할당자는 기본적으로 오버헤드, 내부 단편화, 외부 단편화 3가지 측면에서 메모리를 낭비한다.

메모리 할당자는 모든 자유 블록의 위치, 크기 소유정보, 그리고 내부 상태와 관련된 정보를 부가적으로 저장하고 있다. 이것은 할당자가 관리하는 자체 메모리에 저장이 되며 이는 기존 메모리 할당 규칙을 따라야 한다. 모든 메모리 할당 작업은 프로세서아키텍처에 따라 4,8,16으로 나뉠 수 있는 주소에서 시작해야 한다. 예를 들어 43바이트 블록을 요청 했을 때 44,48또는 그 이상의 바이트를 얻게 되는데 이때 남게 되는 여분의 공간이 바로 내부 단편화이다.

무잠금 메모리풀

무잠금 메모리풀 사용방법

무잠금 메모리 풀을 사용하기 위해서 아래의 헤더를 정의합니다.
#include <MemoryEx.h>

MemoryEx.h 헤더를 열어보면 무잠금 메모리풀을 링크하는 전처리 지시자가 정의되어 있습니다.

```
#pragma comment(lib, "winmm.lib" )
#if defined(_DEBUG)
#pragma comment(lib, "mempooler_x86_mdd.lib")
#else
#pragma comment(lib, "mempooler_x86_md.lib")
#endif
```

또한 헤더파일에는 메모리를 할당/반환하는 함수 및 매크로함수가 선언된 헤더파일이 포함되어 있습니다.
#include <Allocator.h>

아래의 소스는 MyObj 클래스를 무잠금 메모리 풀에서 생성하도록 AllocatorDecl() 매크로 함수를 클래스의 public 맴버로 정의한 예를 보여줍니다.

```
// 무잠금 메모리풀을 사용하려면 반드시 포함해야 합니다.
#include <MemoryEx.h>

class MyObj
{
public:
    MyObj() {}
    virtual ~MyObj() {}     // 디스트럭터 앞에는 반드시 virtual 을 붙여야 한다.

    AllocatorDecl();        // 무잠금 메모리 풀에서  오브젝트를 생성하려면 정의해야 합니다.
};
```

무잠금 메모리풀

```
int _tmain(int argc, _TCHAR* argv[])
{
    MyObj* pObj = new MyObj;  // MyObj 를 new 연산자로 할당하면 무잠금 메모리풀에서 할당한다.

    if ( pObj ) delete pObj;          // 사용이 끝난 MyObj를 삭제하면 메모리가 무잠금 메모리풀에 반환된다.

    return 0;
}
```

무잠금 메모리풀의 성능정보 조회

RPC.NET 은 무잠금 메모리 풀의 메모리 할당 및 반환 등에 대한 상태정보를 실시간으로 조회하는 방법을 제공합니다. 무잠금 메모리 풀의 상태 정보는 mem::get_monitor() 함수를 호출하여 얻어낸 MemoryMonitor 구조체에는 현재 무잠금 메모리풀의 상태 정보를 복사한 복사본이며 구조체의 값을 변경하여도 메모리할당 / 반환 정책에 영향을 주지 않습니다. 아래는 상태정보를 조회하는 방법과 MemoryMonitor 구조체의 맴버들이 어떤 의미를 가지는지 보여줍니다.

```
#include <MemoryEx.h>

int _tmain(int argc, _TCHAR* argv[])
{
    MemPooler_Counter m = performance::mempooler_counter ();
    printf( "사용중인Chunck 개수(m_iInUsed)                    : %lu\n", m.m_iInUsed );
    printf( "Chunck 생성요청횟수(m_iCreated)                   : %lu\n", m.m_iCreated );
    printf( "Chunck 삭제요청횟수(m_iDeleted)                   : %lu\n", m.m_iDeleted );
    printf( "메모리풀에반환된Chunck 개수(m_iStacked)           : %lu\n", m.m_iStacked );
    printf( "할당된총Chunck 크기의합계(m_iBytes)               : %lu\n", m.m_iBytes );
    printf( "메모리풀로부터할당된메모리의전체합계(m_iMaxBytes) : %lu\n", m.m_iMaxBytes );
    printf( "최대Chunck 크기(m_iMaxChunckSize)                 : %lu\n", m.m_iMaxChunckSize );
    printf( "최대Bucket 의개수(m_iMaxBucketCnt)                : %lu\n", m.m_iMaxBucketCnt );
    printf( "메모리풀을재조정하는간격(m_iDurationOfRevision)  : %lu\n", m.m_iDurationOfRevision );

    return 0;
}
```

무잠금 메모리풀

MemPooler의 생성

무잠금 메모리풀은 메모리 할당 및 반환 요청 시 TLS에 MemPooler 오브젝트가 생성되어 있는지 조사하여 MemPooler이 존재하지 않을 경우 MemPooler 를 생성합니다. 때문에 mem::alloc 을 호출하는 쓰레드 별로 개별적인 MemPooler 을 가지게 됩니다. 아래는 MemPooler를 TLS에 생성하는 코드의 일부분입니다.

```
__declspec(thread) MemPooler* m_tlsMemPooler = NULL;
```

MS의 확장 지시어로 TLS에 변수를 선언합니다.

```
void* alloc( size_t iBytes )
{
    if ( m_tlsMemPooler == NULL )
    {
        m_tlsMemPooler = new MemPooler;
    }
    return m_tlsMemPooler->New( iBytes );
}
```

TLS 변수를 조사해서 MemPooler가 생성되지 않았으면 생성합니다.

무잠금 메모리풀에서 메모리를 할당합니다.

```
void free( void* pChunck )
{
    if ( pChunck == NULL ) return;
    if ( m_tlsMemPooler == NULL )
    {
        m_tlsMemPooler = new MemPooler;
    }
    m_tlsMemPooler->Delete( pChunck );
}
```

TLS 변수를 조사해서 MemPooler가 생성되지 않았으면 생성합니다.

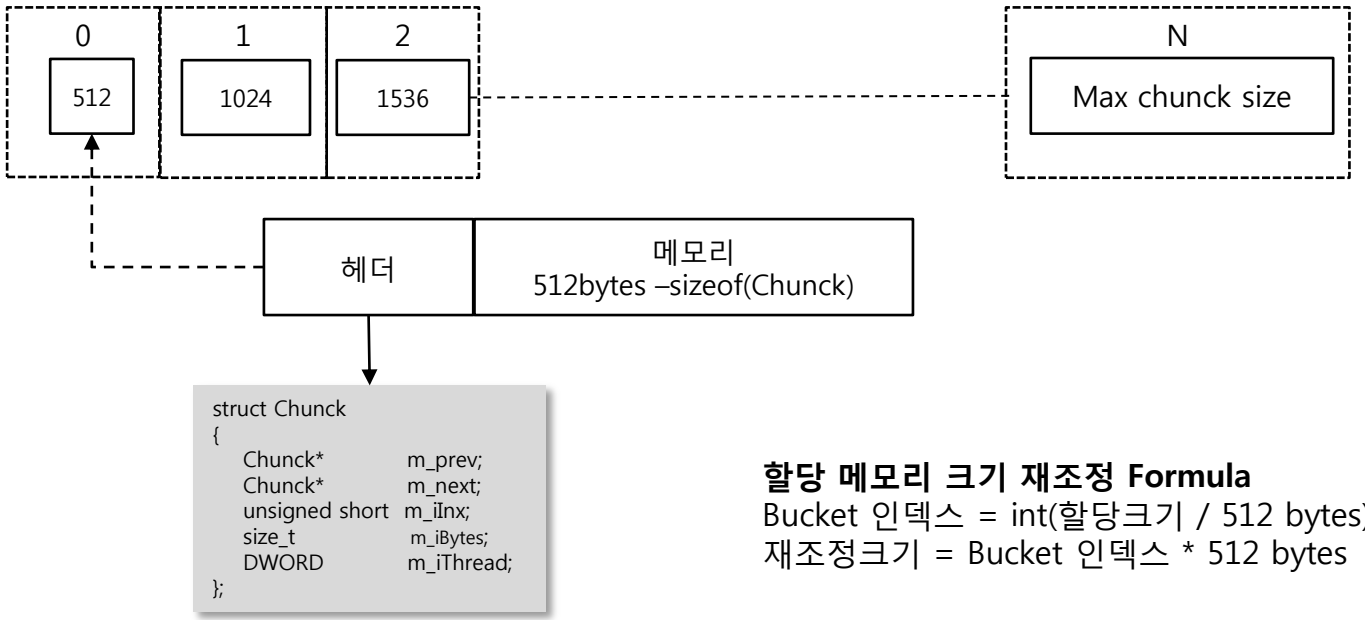
무잠금 메모리풀에서 메모리를 반환합니다.

무잠금 메모리풀

메모리 할당

무잠금 메모리풀에 메모리 할당이 요청되면 무잠금 메모리 풀은 요청한 메모리 크기에 Chunck 헤더 크기를 더한 값이 512 bytes 배수가 되도록 크기를 재조정한 크기로 메모리를 할당한다. 할당된 메모리의 Chunck헤더정보에 는 메모리 반환 시 저장될 bucket의 인덱스 값이 저장된다.
예를 들면 무잠금 메모리풀에 35 bytes의 메모리 할당을 요청하면 "sizeof(Chunck헤더) +35bytes = 50 bytes" 크기를 512 bytes의 배수가 되도록 크기를 재조정하여 512 bytes 메모리를 할당하고 Chunck 헤더의 m_iInx에 0 값을 설정하여 메모리 반환 시 512 bytes 이하의 메모리가 저장되는 0 번째 bucket 에 저장되도록 설정한다. Chunck헤더정보 설정이 완료되면 무잠금 메모리풀은 헤더정보를 제외한 메모리시작주소를 반환한다.

Bucket



할당 메모리 크기 재조정 Formula
Bucket 인덱스 = int(할당크기 / 512 bytes)
재조정크기 = Bucket 인덱스 * 512 bytes

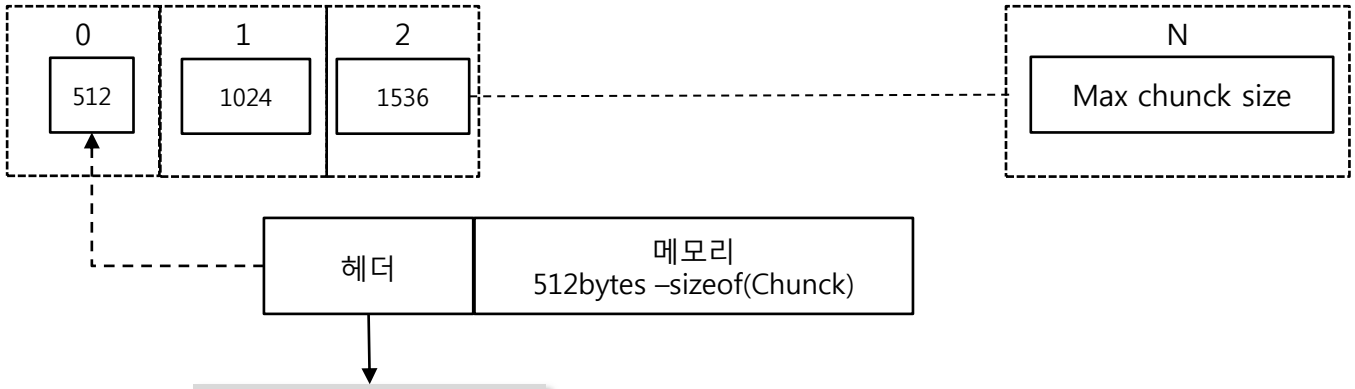
무잠금 메모리풀

메모리 반환

무잠금 메모리풀로 메모리 반환이 요청되면 무잠금 메모리풀은 메모리 할당 시 붙여 놓은 Chunck 헤더 정보를 얻어내기 위해 반환될 메모리의 시작주소에서 Chunck헤더 크기 만큼 주소값을 앞으로 이동한다. Chunck헤더의 m_iInx 에는 반환될 메모리가 저장될 bucket의 인덱스 값이 저장되어 있으며, 이 인덱스 값을 참조하여 bucket의 뒤쪽에 반환할 메모리를 저장한다.

참고)
stl 컨테이너들은 노드를 저장할 때 메모리를 할당하기 때문에 무잠금 메모리풀에서 반환된 메모리를 저장하는 컨테이너로 사용하는 것은 바람직하지 않다. 때문에 무잠금 메모리풀에서는 노드 저장 시점에 별도의 메모리 할당이 필요하지 않은 util::list 를 사용하여 Bucket 을 구현하였다.

Bucket




```
struct Chunck
{
    Chunck*          m_prev;
    Chunck*          m_next;
    unsigned short   m_iInx;
    size_t           m_iBytes;
    DWORD            m_iThread;
};
```

반환된 메모리를 저장할 Bucket 인덱스 계산 Formula
Chunck헤더= 반환메모리주소 – sizeof(Chunck)
Bucket 인덱스 = Chunck헤더->m_iInx

무잠금 메모리풀

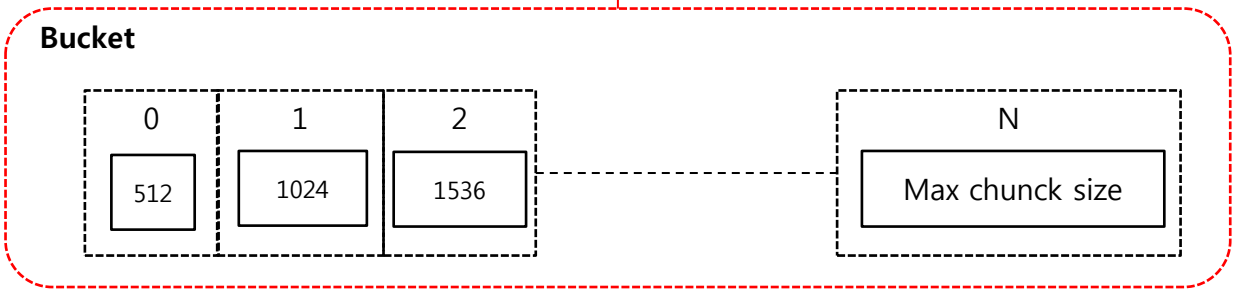
무잠금 메모리풀은 MemPooler 오브젝트를 TLS 에 생성한다. 각 쓰레드 별로 별도의 MemPooler 오브젝트를 참조하여 메모리 할당 및 반환을 수행한다. MemPooler 의 주요 구현를 살펴보면 아래와 같다

```
class MemPooler
{
private:
    struct Chunck : public listnode(Chunck)
    {
        unsigned short          m_iInx;
        size_t                  m_iBytes;
        DWORD                   m_iThread;
    };

    typedef util::list<Chunck>              ChunckList;
    typedef std::vector<ChunckList>         BucketArray;

    BucketArray                             m_Buckets;

};
```

util::list 에 저장할 수 있도록 listnode (Chunck) 를 상속받습니다.

m_Buckets;

반환된 메모리를 크기별로 저장할 Bucket 을 생성합니다.



무잠금 메모리풀

MemPooler의 메모리 할당 및 반환의 구현는 아래와 같습니다.

```
MemPooler::MemPooler( void )
{
    m_Buckets.resize( MONITOR.m_iMaxBucketCnt );
}

void* MemPooler::New( size_t iBytes )
{
    iBytes = iBytes+sizeof(Chunck);
    unsigned short i = Resize(iBytes);

    Chunck* pChk = NULL;
    if ( m_Buckets.size() <= i || m_Buckets[i].size() == 0 )
    {
        pChk = (Chunck*)__malloc(iBytes);
        pChk->m_iInx                = i;
    }
    else
    {
        pChk = m_Buckets[i].front();
        m_Buckets[i].pop_front();
    }
    return ((char*)pChk) + sizeof(Chunck);
}
```

반환된 메모리를 크기별로 저장할 Bucket 을 생성합니다.

요청한 메모리 크기에 Chunck 헤더크기를 더한 값을 512 bytes 배수가 되도록 재조정합니다.

Chunck의 최대 크기를 초과하거나 메모리풀에 반환된 메모리가 없으면 새로운 Chunck를 할당합니다.

메모리풀에 반환된 Chunck가 있으면 꺼냅니다.

Chunck 헤더를 제외한 메모리시작주소를 반환합니다.