Nerv 3.5

소프트웨어 라이브러리 사용자 메뉴얼

(Software Library User Manual)

(Ver 1.0 beta)

2020. 06. 09

(주)웨어그램

목차

<제목 차례>

문서 이력

문서 버전	날짜	작성자	발행번호	비고
1.0 beta	2020-06-10	이용우	wg-20200610-1981-공개	신규발행

문서 요약

Nerv 3.5 매뉴얼

참고 문서

문서 발행번호	발행번호 발급기관	제목	저자

첨부 파일

파일명	내용

용어 정의

본 문서 내에서 다음의 용어들은 별도의 언급이 없는 한 지시한대로의 의미로만 사용합니다.

▶ 분산 시스템

☞ 분산 시스템이란 하나의 Task를 수행할 때에, 여러 개의 노드에 그 데이터와 처리를 분담하게 하는 시스템을 말합니다.

▶ Object

- 데이터(실체)와 그 데이터에 관련되는 동작(절차, 방법, 기능)을 모두 포함한 개념이다.

▶ Nerv

☞ Object와 Object사이의 커뮤니케이션 방식의 한 이름이다.

▶ Nerv Object

☞ Nerv의 방식으로 다른 Object와 커뮤니케이션이 가능한 Object이다.

☞ Nerv 응용프로그램의 런 타임 프로세스이다.

▶ Nerv Object Framework

☞ Nerv의 Object를 제작하는 소스 코드에 대한 프레임워크 방식으로 응용프로그램을 제작하는 것을 말합니다. 이 문서에서는 Nerv와 관련된 모든 것을 통틀어서 Nerv Object Framework라고 부릅니다.

▶ Stub Object

☞ 관계를 맺는 두 Object 사이에서 서비스를 제공하는 Nerv Object이다.

▶ Proxy Object

☞ 관계를 맺는 두 Object 사이에서 서비스를 제공받는 Nerv Object이다.

1. Nerv 3.5 Application 개발 시작하기

1.1. 기본 개념

Nerv 3.5는 기존의 2.4와는 다르게 하나의 라이브러리로만 구성되어 있다. 응용에서는 이 라이브러리의 include 디렉토리를 추가하고, 라이브러리를 링크하여 사용할수 있다. 여기서는 /Nerv35/ 디렉토리가 배포되는 라이브러리 디렉토리의 경로로 가정하고 설명할 것이다. 그리고 Nerv 3.5 라이브러리를 사용시 사용할 C++ 표준은 최소 C++11 이상을 사용할 것을 권장한다.

1.2. 응용 개발 환경 구성

▶ include 디렉토리 추가

☞ 포함 디렉토리(Include Directory)에 다음을 추가한다.

- /Nerv35/include

▶ Library 디렉토리 추가

☞ 라이브러리 디렉토리(Library Directory)에 다음을 추가한다.

- /Nerv35/lib

▶ Library 링크

☞ Windows에서는 응용 프로그램 링크시에 다음 라이브러리를 링크한다.

- /Nerv35/lib/64Nerv35.lib

☞ Linux에서는 응용 프로그램 링크시에 다음 라이브러리를 링크한다.

- /Nerv35/lib/lib64Nerv35.so

▶ 공유 라이브러리 복사

☞ Windows에서는 다음 라이브러리를 C:/Windows/system32 시스템폴더에 복사한다.

- /Nerv35/lib/64Nerv35.dll

☞ Linux에서는 다음 라이브러리를 /usr/lib 라이브러리 디렉토리에 복사한다.

- /Nerv35/lib/lib64Nerv35.so

1.3. 기본 프로그래밍 규칙

1.3.1. Initialize / finalize

▶ Nerv API를 사용하기 전에 먼저 다음을 호출하여야 한다.

☞ Nerv35::initialize()

☞ Console 이라면 main 함수 첫 부분에 작성한다.

☞ MFC라면 Application InitInstance() 함수에서 작성한다.

▶ 프로그램을 종료하기 전에 다음을 호출하여야 한다.

☞ Nerv::finalize()

☞ Console 이라면 main 함수 끝 부분에 작성한다.

☞ MFC라면 Application ExitInstance() 함수에서 작성한다.

▶ 주의사항

☞ Nerv35::initialize() 호출 전에 nerv 객체(nerv_object또는 nerv_proxy_object)의 생성자가 호출될수 없다. 지역변수로 두는 경우 앞에 선언될수 없다.

☞ Nerv35::finalize() 호출 후에 nerv 객체(nerv_object또는 nerv_proxy_object)의 소멸자가 호출될수 없다.

☞ nerv객체를 Nerv35::initialize() Nerv35::finalize() 사이에 선언하는 경우 블록{}으로 생명주기를 제어하여야 한다.

1.3.2. 인터페이스

1.3.2.1. Stub 인터페이스

☞ nerv_object를 virtual 상속 받아 선언하여야 한다.

1.3.2.2. Proxy 인터페이스

☞ nerv_proxy_object를 virtual 상속 받아 선언하여야 한다.

2. Nerv 35 Object Framework

Nerv의 모든 데이터 타입은 namespace Nerv35 안에서 정의된다. 따라서 다른 라이브러리의 타입과 충돌이 일어나더라도 명시적 namespace 기술로서 충돌을 피할 수 있습니다. 다음의 표는 Framework에서 사용하는 헤더 파일이다.

▶ include

헤더파일	설명
nerv_define.hh	호환성 코드 유지를 위한 define
nerv_error.hh	에러 상수 값 정의
nerv_history.hh	nerv_history 클래스 및 history 관련 템플릿 정의
nerv_object_broadcast.hh	nerv_object_broadcast 객체 메서드 클래스 정의
nerv_object_call.hh	nerv_object_call 객체 메서드 클래스 정의
nerv_object_command.hh	nerv_object_command 객체 메서드 클래스 정의
nerv_object_event.hh	nerv_object_event 객체 메서드 클래스 정의
nerv_object_information.hh	nerv_object_information 객체 메서드 클래스 정의
nerv_object_signal.hh	nerv_object_signal 객체 메서드 클래스 정의
nerv_object.hh	nerv_object 객체 클래스 정의
nerv_pre_define.hh	클래스간 의존성을 위한 전방 선언
nerv_proxy_object_broadcast.hh	nerv_proxy_object_broadcast 대리객체 클래스 정의
nerv_proxy_object_call.hh	nerv_proxy_object_call 대리객체 클래스 정의
nerv_proxy_object_command.hh	nerv_proxy_object_command 대리객체 클래스 정의
nerv_proxy_object_event.hh	nerv_proxy_object_event 대리객체 클래스 정의
nerv_proxy_object_information.hh	nerv_proxy_object_information 대리객체 클래스 정의
nerv_proxy_object_signal.hh	nerv_proxy_object_signal 대리객체 클래스 정의
nerv_proxy_object.hh	nerv_proxy_object 대리객체 클래스 정의
nerv_scaled_integer.hh	scaled integer 템플릿 정의
nerv_serial_paraller.hh	메시지 직렬/병렬 처리 제어 함수 정의
nerv_typedef.hh	기본 타입 및 구조체 정의
nerv_utility.hh	유틸리티 함수 정의
Nerv.hh	통합 Include 정의

2.1. 기본 타입

2.1.1. 고정크기 정수 및 부동소수점 타입

☞ #include <Nerv35/nerv_typedef.hh>

타입	추상적 의미	비고
int8_t	8bit 2's complement 정수	Little Endian
int16_t	16bit 2's complement 정수	
int32_t	32bit 2's complement 정수	
int64_t	64bit 2's complement 정수	
uint8_t	8bit 0과 양의 정수	
uint16_t	16bit 0과 양의 정수	
uint32_t	32bit 0과 양의 정수	
uint64_t	64bit 0과 양의 정수	
float32_t	IEEE754 32bit 부동소수점 실수 타입	
float64_t	IEEE754 64bit 부동소수점 실수 타입	

2.1.2. 데이터 크기 및 클록 타입

☞ #include <Nerv35/nerv_typedef.hh>

타입	설명	
pksize_t	의미	32bit 패킷 크기 타입
	기본형 타입	uint32_t
	연관	패킷 데이터의 크기를 지시하는 각종 데이터 및 함수
size_t	의미	32bit 크기 타입
	기본형 타입	uint32_t
	연관	패킷을 제외한 크기를 지시하는 각종 데이터 및 함수
nerv_clock_t	의미	CPU 클럭 카운터 타입
	기본형 타입	u_int64_t
	연관	template_history_elem get_cpu_clock() / get_cpu_hz()

2.1.3. 나열형 데이터 타입

☞ #include <Nerv35/nerv_typedef.hh>

나열형 타입은 Nerv35 라이브러리에서 사용되지 않지만 Standard C++ 11 이상에서 지원하는 고정크기 나열형 타입을 사용할 것을 권장하고, 통합개발환경 자동생성코드에서는 enum class를 활용하여 나열형을 표현할 것이다.

타입	설명	
Sync_model	의미	전송 부분 동기화 모델 타입, command나 information의 경우 post 함수에 동기하여 데이터를 전송할지, 타이머에 의하여 주기적으로 동기할지를 구분하는 모델 타입이다.
	기본형 타입	enum class : uint8_t
	연관	nerv_object_command::on_sync(...) nerv_proxy_object_command::set_sync_model(...) nerv_object_information::set_sync_model(...) nerv_proxy_object_information::on_sync(...) on_post = 0x00 :Post 동기화 방식의 전송모델 on_timer = 0x01 : Timer 주기적 동기화 방식 전송모델

2.1.4. 리턴 Promise 타입

☞ #include <Nerv35/nerv_typedef.hh>

타입	설명	
return_promise_t	의미	리턴 객체 타입, 비동기 리턴을 대기하게하는 약속이다.
	기본형 타입	struct_return_promise*
	연관	nerv_proxy_object_call::async_call(...) nerv_proxy_object_call::async_return(...) nerv_proxy_object_command::async_link(...) nerv_proxy_object_command::async_link_return(...) nerv_proxy_object_event::async_link(...) nerv_proxy_object_event::async_link_return(...) nerv_proxy_object_information::async_link(...) nerv_proxy_object_information::async_link_return(.)

2.1.5. 주소 및 메시지 식별 필드 타입

☞ #include <Nerv35/nerv_typedef.hh>

타입	설명		
msg_code_t	의미	메시지를 식별 하는 타입, 각트랜잭션의 클래스를 생성할 때 고유하게 식별할수 있는 16비트 양수값을 설정한다.	
	기본형 타입	uint16_t	
	연관	nerv_object_call / nerv_object_signal / nerv_object_event / nerv_object_information / nerv_object_command / nerv_object_broadcast / nerv_proxy_object_call / nerv_proxy_object_signal / nerv_proxy_object_event / nerv_proxy_object_information / nerv_proxy_object_command / nerv_proxy_object_datagram	
		각 클래스의 생성자에서 메시지 식별 코드를 지정	
ip_t	의미	컴퓨터 노드 IP주소를 식별하는 타입	
	기본형 타입	32bit 0과 양의 정수	
	연관	nerv_addr_t / server_addr_t / nerv_proxy_object::create(...) inet_addr(...) / inet_ntoa(...)	
port_t	의미	Port주소를 식별하는 타입	
	기본형 타입	uint16_t	
	연관	nerv_addr_t	
nerv_addr_t	의미	Nerv 객체를 식별하는 타입, IP주소와 port주소로 식별한다.	
	struct 멤버	ip	ip_t : 객체의 IP 주소
		port	port_t : 객체의 Port 주소
	연관	nerv_object::on_connect(...) nerv_object::on_disconnect(...)	

2.1.6. 기타 타입

☞ #include <Nerv35/nerv_typedef.hh>

타입	설명	
letter_case_t	의미	Information/Command History 타입, histroy 메모리를 가르키는 포인터 타입이다.
	기본형 타입	struct_letter_case*
	연관	nerv_history::nerv_history(...) nerv_history::operator = (...) template_history::template_history(...) template_history::operator = (...) nerv_???::see_history(...)
nerv_err_t	의미	리턴 에러 타입
	기본형 타입	nerv_const_t
	연관	각종 함수들의 리턴, Nerv의 대부분 함수들은 함수의 처리 결과를 예외값으로써 리턴한다.

2.2. 에러 상수

☞ #include <Nerv35/nerv_error.h>

타입	설명	
nerv_err_t	의미	리턴 에러 타입
	기본형 타입	int32_t
	연관	각종 함수들의 리턴, Nerv의 대부분 함수들은 함수의 처리 결과를 예외값으로써 리턴한다.

▶ 에러리턴값이사용되는곳

- 함수호출트랜잭션의리턴메시지
- 트랜잭션제어리턴메시지(연결응답메시지)

▶ 응용 정의 에러

- 0x00000000~0x7FFFFFFF까지도메인구현에서정의한다.
- 응용에서 정의한 네트워크 통신상에서 주고받는 에러

▶ 통신 에러

- 0x80000000~0xBFFFFFFF
- Nerv35에서 정의한 네트워크 통신상에서 주고받는 에러

▶ 구현 정의 에러

- 0xC0000000~0xFFFFFFFF까지 미들웨어의 구현에서 정의한다.
- 네트워크 통신상으로 주고 받지 않는 에러

2.2.1. 통신 에러

식별자	값	추상적 의미
NERV_ERR_NONE_	0x80000000	함수 실행 성공
NERV_ERR_already_used_name_	0x80000001	이미 사용한 이름
NERV_ERR_OBJECT_IS_NOT_EXIST__	0x80000002	객체가 존재하지 않음
NERV_ERR_not_support_code_	0x80000003	지원하지 않는 메서드 코드
NERV_ERR_not_implement_	0x80000004	원격의 프로그래머에 의하여 구현되지 않음
NERV_ERR_REMOTE_TIMEOUT_	0x80000005	원격 노드에서 메시지의 타임아웃
NERV_ERR_busy_	0x80000006	바쁜 상태로 메시지 처리 실패
NERV_ERR_disconnect_	0x80000007	원격 프로세스가 종료됨

- 원격 노드: 함수 처리 대상 노드
- 로컬 노드: 함수를 호출한 노드(즉, 프로그래밍 중인 노드)

☞ 통신간 에러는 Nerv Protocol 상에서 정의된 기본적인 에러로서 서로 다른 노드상에서 발생할수 있는 기본적인 에러를 나타낸다.

2.2.2. 구현 정의 에러

식별자	값	추상적 의미
NERV_ERR_LOCAL_TIMEOUT__	0xC0000000	로컬 노드에서 메시지의 타임아웃
NERV_ERR_not_connected_tcp_	0xC0000001	tcp가 아직 연결되지 아니함
NERV_ERR_BOUNDOUT_BUFFER__	0xC0000002	output data를 위한 버퍼가 모자람
NERV_ERR_busy_localhost_	0xC0000003	로컬 노드가 바쁜 상태로 메시지 처리 실패
NERV_ERR_TOO_BIG_SIZE__	0xC0000004	지정된 크기보다 데이터 사이즈가 큼
NERV_ERR_IS_NOT_CREATED_	0xC0000005	start 하지 않은 객체
NERV_ERR_NULL_RETURN_PROMISE__	0xC0000006	리턴 객체가 없음
NERV_ERR_already_connected_	0xC0000007	이미 연결되어 있음
NERV_ERR_ALREADY_CREATED__	0xC0000008	이미 start 한 객체
NERV_ERR_disconnected_tcp_	0xC0000009	tcp 연결되지 아니함
NERV_ERR_durty_memory_	0xC000000A	메모리 오염
NERV_ERR_overlaped_packet_	0xC000000B	중복된 패킷

2.2.3. SDK 에러

식별자	값	추상적 의미
NERV_ERR_DONT_GET__	0xD0000001	get 하지 않은 상태에서 post함
NERV_ERR_EMPTY_NERV_MEMORY_	0xD0000002	nerv의 메모리가 모자람
NERV_ERR_empty_process_memory	0xD0000003	프로세스의 힙 메모리가 모자람
NERV_ERR_not_exist_sync_data	0xD0000004	동기화 데이터가 존재하지 않음
NERV_ERR_EMPTY_HISTORY	0xD0000005	History가 비어 있음
NERV_ERR_already_used_addr_	0xD0000006	이미 사용중인 객체 ID
NERV_ERR_not_initailize_	0xD0000007	라이브러리가 초기화 되지 아니함

2.3. Object 클래스

2.3.1. nerv_object

☞ /Nerv35/include/Nerv35/nerv_object.h

```
namespace Nerv35 {
class NERV_SDK_DLL nerv_object {
   friend class Nerv_processor;
   friend class _nerv_object;
public:
   nerv_object();
   virtual ~nerv_object();
   nerv_err_t start(const char* _name_, port_t* _port_);
   nerv_err_t stop();
   bool       is_started();
protected:
   virtual void on_connect(nerv_addr_t _proxy_addr);
   virtual void on_disconnect(nerv_addr_t _proxy_addr);
protected:
   _nerv_object* _op;
};
} // namespace Nerv35
```

☞ Object를 추상화한다.

☞ Stub Object Instance의 최상위 클래스이다.

☞ 상속받을 때 가상 상속(virtual)받아야 한다.

2.3.1.1. 생성자 & 소멸자

Nerv 에서는 생성자와 소멸자 이외에 별도의 start 함수와 destroy 함수를 두어서 라이프 사이클에 대해서 이중적인 구조를 가진다. 이것은 C/C++ 언어의 기본적인 라이프 사이클 이외에 분산처리 환경에서 수시로 반복되는 라이프사이클의 비동기적 환경에 대응하기 위한 것이다.

따라서 실제로 Nerv의 C/C++ 관점의 생성자와 소멸자는 create/destroy 과정을 위한 준비와 정리 동작만을 수행한다.

2.3.1.1.1. nerv_object()

메서드	nerv_object
설명	생성자
Remark	▶ Stub Object로 생성할때의 생성자

2.3.1.1.2. virtual ~nerv_object_call()

메서드	~nerv_object_call
설명	소멸자
Remark	▶ 가상(virtual) 소멸자이다.

2.3.1.2. 메서드

2.3.1.2.1. nerv_err_t start(const char* _name_, port_t* _port_)

메서드	start		
설명	in_domin 도메인의 in_object_id를 노드 내 고유 식별자로 객체를 생성		
파라미터	타입	이름	설명
	c 문자열	_name_	객체의 이름 문자열
	port_t*	_port_	TCP/UDP port를 나타내는 port_t 변수의 주소
리턴	nerv_err_t		생성 처리 결과 에러 코드
	에러		발생원인 / 조치
	NERV_ERR_NONE_		메서드 실행 성공
	NERV_ERR_ALREADY_CREATED_		이미 이 Instance로 start 함 / 로직 수정
	NERV_ERR_EMPTY_NERV_MEMORY_		미들웨어의 메모리가 모자람 / Nerv를 재시작
	NERV_ERR_all_id_is_used		c_any_object_id로 create할 때 여분의 id가 없음 / 이렇게 많은 객체를 생성할 필요가 있는지 검토
	NERV_ERR_already_used_id		이미 사용중인 id를 다시 사용함 / 다른 Instance(다른 Process의 Instance를 포함하여)에서 이미 사용한 ID인지 확인
Remark	▶ _name_ 가 Null이면 이름없는 객체가 된다. ▶ _name_ 가 Null이 아니면 이름 있는 객체가 되고 LAN 상에서 이름을 알린다. ▶ 중복된 _name_ 은 실행은 될수있으나, 의도하지 않은 동작이 발생할 수 있다. 이름은 설계시 유일하게 가질수 있도록 해야 한다. ▶ _port_에는 반드시 존재하는 port_t 변수의 주소를 넘겨야한다. ▶ 그 port_t 변수에 0이 아닌 값이 있으면 해당 값의 port로 생성한다. ▶ 그 port_t 변수에 0의 값이 있으면 임의의 port로 생성하고 그 port_t 변수에 port값을 저장한다.		

2.3.1.2.2. nerv_err_t stop()

메서드	stop	
설명	create한 객체를 더 이상 사용하지 않기 위해서 destroy 함	
리턴	타입	설명
	nerv_err_t	파괴 처리 결과 에러 코드
	에러	발생원인 / 조치
	NERV_ERR_NONE_	메서드 실행 성공
	NERV_ERR_IS_NOT_CREATED_	ceate 하지 않은 Instance를 destroy 함 / 로직 수정
Remark	▶ destroy한 Instance의 다른 method를 더 이상 쓸수 없음 ▶ 다시 쓰기 위해서는 start 하여야 함	

2.3.1.2.3. bool is_started()

메서드	is_started	
설명	create한 Instance 인지 검사함	
리턴	bool	
	값	설명
	true	start 한 Instance
	false	create하지 않았거나, destroy한 Instance

2.3.1.3. 콜백 함수

2.3.1.3.1. virtual void on_connect(node_addr_t _proxy_addr)

함수	on_connect		
설명	Instance를 사용하려는 Proxy 노드와 TCP 세션이 연결되면 호출됨		
파라미터	타입	이름	설명
	node_addr_t	_proxy_addr	Proxy 노드의 주소
Remark	▶ 일반 가상 함수이므로 상속받을 때 가상함수 오버라이딩 하지 않아도 됨 ▶ 필요한 경우 오버라이딩하여 구현한다.		

2.3.1.3.2. virtual void on_disconnect(node_addr_t _proxy_addr)

함수	on_disconnect		
설명	Instance를 사용하려는 Proxy 노드와 TCP 세션이 종료되면 호출됨		
파라미터	타입	이름	설명
	node_addr_t	_proxy_addr	Proxy 노드의 주소
Remark	▶ 일반 가상 함수이므로 상속받을 때 가상함수 오버라이딩 하지 않아도 됨 ▶ 필요한 경우 오버라이딩하여 구현한다.		

2.3.2. nerv_object_call

☞ /Nerv35/include/Nerv35/nerv_object_call.h

```
namespace Nerv35 {
class NERV_SDK_DLL nerv_object_call : virtual public nerv_object {
   friend class _nerv_object_call;
public:
   nerv_object_call(msg_code_t         _code,
                    pksize_t           _max_return_size,
                    const nerv_object& _nerv_object);
   nerv_object_call(msg_code_t _code, pksize_t _max_return_size);
   virtual ~nerv_object_call(void);
private:
   virtual nerv_err_t on_call(nerv_addr_t _proxy_addr,
                              const void* _in_buf_,
                              pksize_t    _in_buf_size,
                              void*       _out_buf_,
                              pksize_t*   _out_buf_size_,
                              double      _elapsed_time,
                              double      _packet_limit_time);
   virtual void on_call_cancel(nerv_addr_t _proxy_addr, nerv_err_t _error);
private:
   _nerv_object_call* _mp;
};
} // namespace Nerv35
```

☞ Object의 Call 트랜잭션을 추상화한다.

☞ Call 트랜잭션 타입의 메서드 클래스는 이 클래스를 일반 상속받아 구현한다.

- 상속받을 때 가상 상속(virtual)받으면 안 된다.
- 가상 상속 받으면 Call Method는 오직하나만 존재할 수밖에 없기 때문이다.

2.3.2.1. 생성자 & 소멸자

2.3.2.1.1. nerv_object_call(msg_code_t _code, pksize_t _max_return_size)

메서드	nerv_object_call		
설명	리턴 데이터가 _max_return_size로 제한되고, _code로 식별되는 Call 트랜잭션 생성		
파라미터	타입	이름	설명
	msg_code_t	_code	Call 식별 코드
	pksize_t	_max_return_size	리턴메시지의 최대 크기
Remark	▶ 여기에서 _code는 Call의 의미에 대한 식별 코드값이다. 예를 들면 _code=0x0001 은 printf 함수로, _code=0x0002는 scanf 함수로 의미를 결정할 수 있습니다. ▶ 제약사항 ☞ _code = 0x0000은 사용할 수 없습니다. ☞ _max_return_size는 환경 설정에 따라 제약될 수 있다.		

2.3.2.1.2. virtual ~nerv_object_call()

메서드	~nerv_object_call
설명	소멸자
Remark	☞ 가상(virtual) 소멸자이다.

2.3.2.2. 콜백 함수

2.3.2.2.1. virtual nerv_err_t on_call(...)

함수	on_call		
설명	proxy 로부터 Call 요청을 수신하면 호출		
파라미터	타입	이름	설명
	nerv_addr_t	_proxy_addr	Proxy 객체의 주소
	const void*	_in_buf_	Input Data 버퍼의 포인터
	pksize_t	_in_buf_size	Input Data 버퍼 크기
	void*	_out_buf_	Output Data 버퍼의 포인터
	pksize_t*	_out_buf_size_	Output Data의 크기 포인터 전달
	double	_elapsed_time	메서드 호출 진행 시간
	double	_packet_limit_time	메서드 제한 시간
리턴	nerv_err_t		처리 결과 에러 코드값
Remark	▶ Output Data 버퍼는 생성자의 in_max_return_size 크기에 제한됨 ▶ Call은 on_call 함수가 먼저 호출된 후에 리턴을 proxy에 전송함 ▶ 함수 처리가 정상적인 경우 NERV_ERR_NONE_(0x00000000)을 리턴 ▶ 각 함수의 에러값은 0x00000000~0x7FFFFFFF 범위에서 정의 ▶ 에러코드의 값은 각 함수코드마다 그 의미가 다를 수 있음		

2.3.2.2.2. virtual void on_call_cancel(_nerv_addr_t _proxy_addr, nerv_err_t _error)

함수	on_call_cancel		
설명	call 호출이 취소됨		
파라미터	타입	이름	설명
	nerv_addr_t	_proxy_addr	Proxy 노드의 주소
	erv_err_t	_error	에러값
Remark	▶ cancel 동작을 구현하지 않더라도 오버라이딩하여 구현내용이 없음을 명시함 ▶ on_call에서 NERV_ERR_NONE_을 리턴한 경우 아래와 같은 에러에 의하여 이 함수가 호출될 수 있음 ▶ on_call에서 NERV_ERR_NONE_아 아닌 값을 리턴한 경우 이 함수는 호출되지 아니함		
에러	에러	발생원인 / 조치	
	NERV_ERR_REMOTE_TIMEOUT_	proxy 측 노드에서 리턴 패킷이 타임아웃	
	NERV_ERR_OBJECT_IS_NOT_EXIST_	proxy 객체가 destroy되어 리턴 전달 불가	
	NERV_ERR_LOCAL_TIMEOUT_	stub 측 노드에서 리턴 패킷이 타임아웃	
	기타	proxy에서 직접 call_cancel 함수를 호출	

2.3.3. nerv_object_signal

☞ /Nerv35/include/Nerv35/nerv_object_signal.h

```
namespace Nerv35 {
class NERV_SDK_DLL nerv_object_signal : virtual public nerv_object {
   friend class _nerv_object_signal;
public:
   nerv_object_signal(msg_code_t _code);
   virtual ~nerv_object_signal(void);
private:
   virtual void on_signal(nerv_addr_t _proxy_addr,
                          const void* _buf_,
                          pksize_t    _buf_size,
                          double      _elapsed_time);
   virtual void on_signal_cancel(nerv_addr_t _proxy_addr, nerv_err_t _error);
private:
   _nerv_object_signal* _mp;
};
} // namespace Nerv35
```

☞ Object의 Signal 트랜잭션을 추상화한다.

☞ Signal 트랜잭션 타입의 메서드 클래스는 이 클래스를 일반 상속받아 구현한다.

- 상속받을 때 가상 상속(virtual)받으면 안 된다.
- 가상 상속 받으면 Signal Method는 오직하나만 존재할 수밖에 없기 때문이다.

2.3.3.1. 생성자 & 소멸자

2.3.3.1.1. nerv_object_signal(msg_code_t _code)

메서드	nerv_object_signal		
설명	_code로 식별되는 Signal 트랜잭션을 생성합니다.		
파라미터	타입	이름	설명
	msg_code_t	_code	Signal 식별 코드
Remark	▶ 여기에서 _code는 시그널의 의미에 대한 식별 코드 값이다. 예를 들면 code=0x0001은 키보드 타이핑 시그널, code=0x0002는 마우스 시그널과 같이 구분하는데 사용 된다. ▶ 제약사항 ☞ _code = 0x0000은 사용할 수 없습니다.		

2.3.3.1.2. virtual ~nerv_object_signal()

메서드	~nerv_object_signal
설명	소멸자
Remark	☞ 가상(virtual) 소멸자이다.

2.3.3.2. 콜백 함수

2.3.3.2.1. virtual void on_signal(nerv_addr_t, const void*, pksize_t, double)

함수	on_signal		
설명	proxy 로부터 Signal 요청을 수신하면 호출		
파라미터	타입	이름	설명
	nerv_addr_t&	_proxy_addr	Proxy 객체의 주소
	void* const	_buf_	Input Data 버퍼의 포인터
	pksize_t	_buf_size	Input Data 버퍼 크기
	double	_elapsed_time	메서드 호출 진행 시간
Remark	▶ on_call과 다르게 on_signal 호출에는 제한시간이 없다. ▶ Signal의 리턴메시지를 먼저 Proxy에 전송한 후에 on_signal 함수가 호출됨 ☞ Call은 on_call 함수가 먼저 호출된 후에 리턴을 proxy에 전송함 ☞ signal도 Call처럼 전달성을 리턴 메시지로 검증한다. 단, on_signal 처리 이후가 아닌 on_signal 처리 전에 리턴 메시지를 전송한다.		

2.3.3.2.2. virtual void on_signal_cancel(nerv_addr_t, nerv_err_t) = 0

함수	on_signal_cancel		
설명	signal 호출이 취소됨		
파라미터	타입	이름	설명
	nerv_addr_t	_proxy_addr	Proxy 노드의 주소
	nerv_err_t	_error	에러값
Remark	▶ 순수 가상함수이므로 인스턴스화 하기 전에 반드시 구현하여야 함 ▶ cancel 동작을 구현하지 않더라도 오버라이딩하여 구현내용이 없음을 명시함 ▶ 아래와 같은 에러에 의하여 이 함수가 호출될 수 있음		
에러	에러	발생원인 / 조치	
	NERV_ERR_REMOTE_TIMEOUT_	proxy 측 노드에서 리턴 패킷이 타임아웃	
	NERV_ERR_OBJECT_IS_NOT_EXIST__	proxy 객체가 destroy되어 리턴 전달 불가	
	NERV_ERR_LOCAL_TIMEOUT__	stub 측 노드에서 리턴 패킷이 타임아웃	
	기타	proxy에서 직접 call_cancel 함수를 호출	

2.3.4. nerv_object_event

☞ /Nerv35/include/Nerv35/nerv_object_event.h

```
namespace Nerv35 {
class NERV_SDK_DLL nerv_object_event : virtual public nerv_object {
   friend class _nerv_object_event;
public:
   nerv_object_event(msg_code_t _code);
   nerv_object_event(msg_code_t _code, const nerv_object& __nerv_object);
   virtual ~nerv_object_event();
   void       unlink_all_proxy();
   void       unlink(nerv_addr_t _proxy_addr);
   nerv_err_t event(const void* _buffer_, pksize_t _size);
   nerv_err_t event();
private:
   virtual nerv_err_t on_link(nerv_addr_t _proxy_addr,
                              double      _packet_live_time,
                              double      _packet_limit_time);
   virtual void       on_unlink(nerv_addr_t _proxy_addr, nerv_err_t _err);
private:
   _nerv_object_event* _mp;
};
} // namespace Nerv35
```

☞ Object의 Event 트랜잭션을 추상화한다.

☞ Event 트랜잭션 타입의 메서드 클래스는 이 클래스를 일반 상속받아 구현한다.

- 상속받을 때 가상 상속(virtual)받으면 안 된다.
- 가상 상속 받으면 Event Method는 오직하나만 존재할 수밖에 없기 때문이다.

2.3.4.1. 생성자 & 소멸자

2.3.4.1.1. nerv_object_event(msg_code_t _code)

메서드	nerv_object_event		
설명	_code로 식별되는 Event 트랜잭션을 생성합니다.		
파라미터	타입	이름	설명
	msg_code_t	_code	Event 식별 코드
Remark	▶ 여기에서 _code는 이벤트의 의미에 대한 식별 코드 값이다. 예를 들면 code=0x0001은 키보드 타이핑 이벤트, code=0x0002는 마우스 이벤트과 같이 구분하는데 사용 된다. ▶ 제약사항 ☞ _code = 0x0000은 사용할 수 없습니다.		

2.3.4.1.2. virtual ~nerv_object_event()

메서드	~nerv_object_event
설명	소멸자
Remark	☞ 가상(virtual) 소멸자이다.

2.3.4.2. 메서드

2.3.4.2.1. void unlink(nerv_addr_t _proxy_addr)

메서드	unlink		
설명	_proxy_addr 주소의 Proxy 객체와의 이벤트 연결을 종료한다.		
파라미터	타입	이름	설명
	nerv_addr_t	_proxy_addr	proxy 객체의 주소
Remark	▶ 지시한 객체와의 연결을 종료하기 원할 때 사용한다. ▶ 이후에 해당 객체는 Event Notification 대상에서 제외된다.		

2.3.4.2.2. void unlink_all_proxy()

메서드	unlink_all_proxy
설명	모든 Proxy 객체와의 이벤트 연결을 종료한다.
Remark	▶ 모든 Proxy 객체와의 연결을 종료하기 원할 때 사용한다. ▶ 이후에 Event Notification는 어느 객체에도 전달되지 않는다.

2.3.4.2.3. void event(const void* _buffer_, pksize_t _size)

메서드	event		
설명	데이터를 실어서 이벤트를 통지한다.		
파라미터	타입	이름	설명
	const void*	_buffer_	데이터의 버퍼
	pksize_t	_size	데이터의 크기
Remark	▶ 데이터와 함께 이벤트를 통지하는 경우에 한하여 사용 ▶ link가 성립된 Proxy 객체에 대해서만 이벤트가 통지됨		

2.3.4.2.4. void event()

메서드	event
설명	데이터 없이 이벤트만 통지한다.
Remark	▶ 데이터 없이 이벤트를 통지하는 경우에 한하여 사용한다. ▶ link가 성립된 Proxy 객체에 대해서만 이벤트가 통지됨

2.3.4.3. 콜백 함수

2.3.4.3.1. virtual nerv_err_t on_link(nerv_addr_t, double, double)

함수	on_link		
설명	proxy 로부터 Event link 요청을 수신하면 호출		
파라미터	타입	이름	설명
	nerv_addr_t	_proxy_addr	Proxy 객체의 주소
	double	_packet_live_time	link 요청 진행 시간
	double	_packet_limit_time	link 요청 제한 시간
리턴	nerv_err_t		연결 지시
Remark	▶ Event는 on_link 함수가 먼저 호출된 후에 confirm을 proxy에 전송함 ▶ 연결을 정상적으로 수립시키려면 NERV_ERR_NONE_(0x00000000)을 리턴 ▶ 각 Event link의 에러값은 0x00000000~0x7FFFFFFF 범위에서 정의 ☞ 에러코드의 값은 각 Event 마다 그 의미가 다를 수 있음		

2.3.4.3.2. virtual void on_unlink(nerv_addr_t , nerv_err_t)

함수	on_unlink		
설명	link 연결이 Proxy로부터 종료됨		
파라미터	타입	이름	설명
	nerv_addr_t	_proxy_addr	Proxy 노드의 주소
	nerv_err_t	_err	원인 에러
Remark	▶ 해당 proxy 객체와 link 상태에서 연결이 끊길 때 호출됨 ▶ 다음과 같은 원인으로 on_unlink 이 호출되어질 수 있음 ☞ proxy 객체에서 unlink를 호출 ☞ proxy 객체가 destroy됨 ☞ proxy 객체의 process 가 종료됨(비정상 종료 포함) ☞ 네트워크 불안정으로 TCP 세션이 종료됨 ▪ hearbit 통신이 일정시간 이상 교류되지 아니함 ☞ proxy 객체의 노드를 종료함(비정상 종료 포함)		

2.3.5. nerv_object_information

☞ /Nerv35/include/Nerv35/nerv_object_information.h

```
namespace Nerv35 {
class NERV_SDK_DLL nerv_object_information : virtual public nerv_object {
   friend class _nerv_object_information;
public:
   nerv_object_information(msg_code_t _code);
   nerv_object_information(msg_code_t _code, const nerv_object& __nerv_object);
   virtual ~nerv_object_information(void);
   void          set_sync_model(Sync_model _sync_model, double _interval);
   void          unlink(nerv_addr_t _proxy_addr);
   void          unlink_all_proxy();
   void*         get(pksize_t _size);
   void          cancel();
   nerv_err_t    post(pksize_t _size);
   nerv_err_t    write(const void* _data_, pksize_t _size);
   letter_case_t see_history(size_t _size);
   void          set_history_size(size_t _history_size);
   nerv_err_t    read(void* _data_, pksize_t* _size_);
private:
   virtual nerv_err_t on_link(nerv_addr_t _proxy_addr,
                              bool        _reliability,
                              double      _packet_live_time,
                              double      _packet_limit_time);
   virtual void      on_unlink(nerv_addr_t _proxy_addr, nerv_err_t _err);
private:
   _nerv_object_information* _mp;
};
} // namespace Nerv35
```

☞ Object의 Information 트랜잭션을 추상화한다.

☞ Information 트랜잭션 타입의 메서드 클래스는 이 클래스를 일반 상속받아 구현한다.

- 상속받을 때 가상 상속(virtual)받으면 안 된다.
- 가상 상속 받으면 Information Method는 오직하나만 존재할 수밖에 없기 때문이다.

2.3.5.1. 생성자 & 소멸자

2.3.5.1.1. nerv_object_information(msg_code_t _code)

메서드	nerv_object_information		
설명	_code로 식별되는 Information 트랜잭션을 생성합니다.		
파라미터	타입	이름	설명
	msg_code_t	_code	Information 식별 코드
Remark	▶ 여기에서 _code는 Information의 의미에 대한 식별 코드값이다. 예를 들면 code=0x0001은 Sound 정보, code=0x0002는 Image 정보와 같이 구분하는데 사용 된다. ▶ 제약사항 ☞ _code = 0x0000은 사용할 수 없습니다.		

2.3.5.1.2. virtual ~nerv_object_information()

메서드	~nerv_object_information
설명	소멸자
Remark	☞ 가상(virtual) 소멸자이다.

2.3.5.2. 메서드

2.3.5.2.1. void set_sync_model(Sync_model , double)

메서드	set_sync_model		
설명	information의 데이터 동기 방식을 선택합니다.		
파라미터	타입	이름	설명
	Sync_model	_sync_model	동기화 방식
	double	_interval	시간 간격
Remark	▶ _sync_model은 다음 두 개의 값중 하나 ☞ Sync_model::on_post ▪ post / write 함수를 호출하는 타이밍에 데이터를 동기화 ▪ 이때 _interval 은 패킷 제한 시간 ☞ Sync_model::on_timer ▪ Nerv 내부 타이머에 의하여 데이터를 주기적으로 동기화 ▪ 이때 _interval은 동기화 시간 간격		

2.3.5.2.2. void unlink(nerv_addr_t _proxy_addr)

메서드	unlink		
설명	_proxy_addr 주소의 Proxy 객체와의 이벤트 연결을 종료한다.		
파라미터	타입	이름	설명
	nerv_addr_t	_proxy_addr	proxy 객체의 주소
Remark	▶ 지시한 객체와의 연결을 종료하기 원할 때 사용한다. ▶ 이후에 해당 객체는 Information 동기화 대상에서 제외된다.		

2.3.5.2.3. void unlink_all_proxy()

메서드	unlink_all_proxy		
설명	모든 Proxy 객체와의 이벤트 연결을 종료한다.		
Remark	▶ 모든 Proxy 객체와의 연결을 종료하기 원할 때 사용한다. ▶ 이후에 해당 객체는 Information 동기화 대상에서 제외된다.		

2.3.5.2.4. void* get(pksize_t _size)

메서드	get		
설명	동기화 할 데이터를 작성할 메모리를 가져온다.		
파라미터	타입	이름	설명
	pksize_t	_size	가져올 메모리의 크기
리턴	void*		메모리 주소
Remark	▶ information이나 command의 데이터 작성 함수 ☞ get/post(cancel) 방식 ▪ get 이후에는 반드시 cancel 로 취소하거나 post로 작성을 완료하여야 한다. ▪ Nerv의 메모리를 직접 획득하므로 데이터 카피가 발생하지 않는다. ☞ write 방식 ▪ get/post(cancel)방식은 아무래도 3개의 함수로 작성하기 때문에 코드가 복잡해 보일수 있어서 데이터의 크기 때문에 문제가 없는 것은 가독성을 위해서 write함수를 사용한다. ▪ 내부적으로 get/post(cancel) 과 memcpy 함수를 이용한다. ▪ 데이터 크기가 크면 유저 메모리에서 Nerv메모리로 복사하는 과정을 가지므로 속도 저하의 원인이 될 수도 있다. ▶ Nerv 메모리가 모자라는 경우 NULL 을 리턴한다. ▪ 0을 리턴한 경우 post/cancel 하지 않는다.		

2.3.5.2.5. void cancel()

메서드	cancel		
설명	get으로 가져온 메모리를 취소한다.		

2.3.5.2.6. nerv_err_t post(pksize_t _size)

메서드	post		
설명	get으로 가져온 메모리에 데이터 작성을 완료한다.		
파라미터	타입	이름	설명
	pksize_t	_size	작성 완료한 데이터의 크기
리턴	nerv_err_t		
	에러	발생원인 / 조치	
	NERV_ERR_NONE_	메서드 실행 성공	
	NERV_ERR_IS_NOT_CREATED_	객체를 create하지 않음	
	NERV_ERR_DONT_GET__	get으로 메모리를 확보하지 않음	
	NERV_ERR_TOO_BIG_SIZE__	get으로 확보한 메모리보다 큰 크기로 post함	

2.3.5.2.7. nerv_err_t write(const void* _data_, pksize_t _size)

메서드	write		
설명	get으로 가져온 메모리에 데이터 작성을 완료한다.		
파라미터	타입	이름	설명
	const void*	_data_	작성할 원본 데이터 버퍼
	pksize_t	_size	작성할 원본 데이터 크기
리턴	nerv_err_t		
	에러	발생원인 / 조치	
	NERV_ERR_NONE_	메서드 실행 성공	
	NERV_ERR_IS_NOT_CREATED_	객체를 create하지 않음	
	NERV_ERR_EMPTY_NERV_MEMORY_	Nerv 메모리가 모자람	
Remark	▶ void* get(pksize_t _size) 참조		

2.3.5.2.8. letter_case_t see_history(size_t _size)

메서드	see_history		
설명	post/write 한 데이터의 History 기록을 가져온다.		
파라미터	타입	이름	설명
	size_t	_size	가져올 데이터의 최대 개수
리턴	letter_case_t		DLL history의 포인터
Remark	▶ template_history의 생성자에 리턴을 넣어 사용한다. ▶ 배열형으로써 첨자 [] 방식으로 데이터를 접근할 수 있다. [n] 데이터가 [n+1]보다 최근이고, [0]은 가장 최근 데이터이다. ▶ set_history_size()를 통해서 최대 보관 크기를 지시할 수 있다. ▶ start 하지 않은 경우 NULL 포인터를 리턴한다.		

2.3.5.2.9. void set_history_size(size_t _history_size)

메서드	set_history_size		
설명	History의 최대 보관 개수를 지시한다.		
파라미터	타입	이름	설명
	size_t	_history_size	보관할 데이터의 최대 개수

2.3.5.2.10. nerv_err_t read(void* _data_, pksize_t* p_size)

메서드	read		
설명	가장 최근에 작성한 데이터를 읽어온다.		
파라미터	타입	이름	설명
	void*	_data_	읽어올 데이터 버퍼
	pksize_t*	_size_	읽어올 데이터 크기
리턴	nerv_err_t		
	에러		발생원인 / 조치
	NERV_ERR_NONE_		메서드 실행 성공
	NERV_ERR_IS_NOT_CREATED_		객체를 create하지 않음
	NERV_ERR_EMPTY_HISTORY		읽어올 데이터가 없음

2.3.5.3. 콜백 함수

2.3.5.3.1. virtual nerv_err_t on_link(nerv_addr_t, bool, double, double)

함수	on_link		
설명	proxy 로부터 Event link 요청을 수신하면 호출		
파라미터	타입	이름	설명
	nerv_addr_t	_proxy_addr	Proxy 객체의 주소
	bool	_reliability	TCP 전송방식 true / UDP false
	double	_packet_live_time	link 요청 진행 시간
	double	_packet_limit_time	link 요청 제한 시간
리턴	nerv_err_t		
Remark	▶ 순수 가상함수이므로 인스턴스화 하기 전에 반드시 구현하여야 함 ▶ Event는 on_link 함수가 먼저 호출된 후에 confirm을 proxy에 전송함 ▶ 연결을 정상적으로 수립시키려면 NERV_ERR_NONE_(0x00000000)을 리턴 ▶ 각 Information link의 에러값은 0x80000000~0xFFFFFFFF 범위에서 정의 ☞ 에러코드의 값은 각 Event 마다 그 의미가 다를 수 있음		

2.3.5.3.2. virtual void on_unlink(nerv_addr_t _proxy_addr, nerv_err_t _err)

함수	on_unlink		
설명	link 연결이 Proxy로부터 종료됨		
파라미터	타입	이름	설명
	nerv_addr_t	_proxy_addr	Proxy 노드의 주소
	nerv_err_t	_err	연결종료 원인 에러
Remark	▶ 순수 가상함수이므로 인스턴스화 하기 전에 반드시 구현하여야 함 ▶ unlink 동작을 구현하지 않더라도 오버라이딩하여 구현내용이 없음을 명시함 ▶ 해당 proxy 객체와 link 상태에 있어야 on_unlink이 호출되어짐 ▶ 다음과 같은 원인으로 on_unlink 이 호출되어질 수 있음 ☞ proxy 객체에서 unlink를 호출 ☞ proxy 객체가 destroy됨 ☞ proxy 객체의 process 가 종료됨(비정상 종료 포함) ☞ 네트워크 불안정으로 TCP 세션이 종료됨 ▪ hearbit 통신이 일정시간 이상 교류되지 아니함 ☞ proxy 객체의 노드를 종료함(비정상 종료 포함)		

2.3.6. nerv_object_command

☞ /Nerv35/include/Nerv35/nerv_object_command.h

```
namespace Nerv35 {
class NERV_SDK_DLL nerv_object_command : public virtual nerv_object {
   friend class _nerv_object_command;
public:
   nerv_object_command(msg_code_t _code);
   nerv_object_command(msg_code_t _code, const nerv_object& __nerv_object);
   virtual ~nerv_object_command();
   void          unlink(nerv_addr_t _proxy_addr);
   void          unlink_all_proxy();
   void          set_sync(bool _use_on_sync);
   letter_case_t see_history(nerv_addr_t _proxy_addr, size_t _size);
   void          set_history_size(size_t _history_size);
   nerv_err_t    read(nerv_addr_t _client, void* _data_, pksize_t* _size_);
private:
   virtual void      on_sync(nerv_addr_t _proxy_addr,
                             Sync_model  _sync_model,
                             double      _elapsed_time,
                             double      _limit,
                             const void* _buffer_,
                             pksize_t    _size);
   virtual nerv_err_t on_link(nerv_addr_t _client,
                             double      _packet_live_time,
                             double      _packet_limit_time);
   virtual void      on_unlink(nerv_addr_t _client, nerv_err_t _err);
private:
   _nerv_object_command* _mp;
};
} // namespace Nerv35
```

☞ Object의 Command 트랜잭션을 추상화한다.

☞ Command 트랜잭션 타입의 메서드 클래스는 이 클래스를 일반 상속받아 구현한다.

- 상속받을 때 가상 상속(virtual)받으면 안 된다.
- 가상 상속 받으면 Command Method는 오직하나만 존재할 수밖에 없기 때문이다.

2.3.6.1. 생성자 & 소멸자

2.3.6.1.1. nerv_object_command(msg_code_t _code)

메서드	nerv_object_command		
설명	_code로 식별되는 Command 트랜잭션을 생성합니다.		
파라미터	타입	이름	설명
	msg_code_t	_code	Command 식별 코드
Remark	▶ 여기에서 _code는 Command의 의미에 대한 식별 코드값이다. 예를 들면 code=0x0001은 조이스틱 핸들 명령, code=0x0002는 조이스틱 버튼 명령와 같이 구분하는데 사용 된다. ▶ 제약사항 ☞ _code = 0x0000은 사용할 수 없습니다.		

2.3.6.1.2. virtual ~nerv_object_command()

메서드	~nerv_object_command
설명	소멸자
Remark	☞ 가상(virtual) 소멸자이다.

2.3.6.2. 메서드

2.3.6.2.1. void unlink(nerv_addr_t _proxy_addr)

메서드	unlink		
설명	_proxy_addr 주소의 Proxy 객체와의 Command 연결을 종료한다.		
파라미터	타입	이름	설명
	nerv_addr_t	_proxy_addr	proxy 객체의 주소
Remark	▶ 지시한 객체와의 연결을 종료하기 원할 때 사용한다. ▶ 이후에 해당 객체는 Command 동기화 대상에서 제외된다.		

2.3.6.2.2. void unlink_all_proxy()

메서드	unlink_all_proxy
설명	모든 Proxy 객체와의 이벤트 연결을 종료한다.
Remark	▶ 모든 Proxy 객체와의 연결을 종료하기 원할 때 사용한다. ▶ 이후에 어떤 객체로부터도 동기화되지 않는다.

2.3.6.2.3. void set_sync(bool _use_on_sync)

메서드	set_sync		
설명	on_sync 함수를 호출해 줄지 지시한다.		
파라미터	타입	이름	설명
	bool	_use_on_sync	호출 true / 비호출 false
Remark	▶ 지시하지 않는 경우 기본적으로 호출 true 이다.		

2.3.6.2.4. letter_case_t see_history(nerv_addr_t , size_t)

메서드	see_history		
설명	동기화 된 데이터의 History 기록을 가져온다.		
파라미터	타입	이름	설명
	nerv_addr_t	_proxy_addr	읽어올 History의 Proxy 객체 주소
	size_t	_size	가져올 데이터의 최대 개수
리턴	letter_case_t		DLL history의 포인터
Remark	▶ template_history의 생성자에 리턴을 넣어 사용한다. ▶ 배열형으로써 첨자 [] 방식으로 데이터를 접근할 수 있다. [n] 데이터가 [n+1]보다 최근이고, [0]은 가장 최근 데이터이다. ▶ set_history_size()를 통해서 최대 보관 크기를 지시할 수 있다. ▶ start 하지 않았거나 _proxy_addr와 link 되지 않은 경우 NULL 포인터를 리턴한다.		

2.3.6.2.5. void set_history_size(size_t _history_size)

메서드	set_history_size		
설명	History의 최대 보관 개수를 지시한다.		
파라미터	타입	이름	설명
	size_t	_history_size	보관할 데이터의 최대 개수

2.3.6.2.6. nerv_err_t read(nerv_addr_t, void*, pksize_t*)

메서드	read		
설명	가장 최근에 동기화 된 데이터를 읽어온다.		
파라미터	타입	이름	설명
	nerv_addr_t	_proxy_addr	읽어올 데이터의 Proxy 객체 주소
	void*	_data_	읽어올 데이터 버퍼
	pksize_t*	_size_	읽어올 데이터 크기
리턴	nerv_err_t		
	에러	발생원인 / 조치	
	NERV_ERR_NONE_	메서드 실행 성공	
	NERV_ERR_IS_NOT_CREATED_	객체를 create하지 않음	
	NERV_ERR_EMPTY_HISTORY	link 되지 않았거나, 동기화된 데이터가 없음	

2.3.6.3. 콜백 함수

2.3.6.3.1. virtual void on_sync(...)

함수	on_sync		
설명	proxy 로부터 Command Data가 동기화 직후 호출됨		
파라미터	타입	이름	설명
	nerv_addr_t	_proxy_addr	Proxy 객체의 주소
	Sync_model	_sync_model	동기화 모델
	double	_elapsed_time	동기화 진행 시간
	double	in_limit	동기화 제한 시간
	const void*	_buffer_	동기화 데이터
	const pksize_t	_size	동기화 데이터 크기
Remark	▶ 순수 가상함수이므로 인스턴스화 하기 전에 반드시 구현하여야 함 ▶ 동기화 직후(read 함수로 읽을 수 있는 상태)에 호출됨 ▶ 동기화 데이터를 이 함수 외부에서 읽기 위해서는 read 함수를 사용 ▶ 모든 동기화 데이터를 개별적으로 식별하고, 정확한 동기화 타이밍에 어떤 작업을 하기위해서는 이 함수에서 구현해야 한다.		

2.3.6.3.2. virtual nerv_err_t on_link(...)

함수	on_link		
설명	proxy 로부터 Command link 요청을 수신하면 호출		
파라미터	타입	이름	설명
	nerv_addr_t	_proxy_addr	Proxy 객체의 주소
	bool	_tcp_request	TCP 전송방식 true / UDP false
	double	_packet_live_time	link 요청 진행 시간
	double	_packet_limit_time	link 요청 제한 시간
리턴	nerv_err_t		
Remark	▶ 순수 가상함수이므로 인스턴스화 하기 전에 반드시 구현하여야 함 ▶ Command는 on_link 함수가 먼저 호출된 후에 confirm을 proxy에 전송함 ▶ 연결을 정상적으로 수립시키려면 NERV_ERR_NONE_(0x00000000)을 리턴 ▶ 각 Command link의 에러값은 0x80000000~0xFFFFFFFF 범위에서 정의 ☞ 에러코드의 값은 각 Event 마다 그 의미가 다를 수 있음		

2.3.6.3.3. virtual void on_unlink(nerv_addr_t)

함수	on_unlink		
설명	link 연결이 Proxy로부터 종료됨		
파라미터	타입	이름	설명
	nerv_addr_t	_proxy_addr	Proxy 노드의 주소
Remark	▶ 순수 가상함수이므로 인스턴스화 하기 전에 반드시 구현하여야 함 ▶ unlink 동작을 구현하지 않더라도 오버라이딩하여 구현내용이 없음을 명시함 ▶ 해당 proxy 객체와 link 상태에 있어야 on_unlink이 호출되어짐 ▶ 다음과 같은 원인으로 on_unlink 이 호출되어질 수 있음 ☞ proxy 객체에서 unlink를 호출 ☞ proxy 객체가 destroy됨 ☞ proxy 객체의 process 가 종료됨(비정상 종료 포함) ☞ 네트워크 불안정으로 TCP 세션이 종료됨 ▪ hearbit 통신이 일정시간 이상 교류되지 아니함 ☞ proxy 객체의 노드를 종료함(비정상 종료 포함)		

2.3.7. nerv_object_broadcast

☞ /Nerv35/include/Nerv35/nerv_object_broadcast.h

```
namespace Nerv35 {
class NERV_SDK_DLL nerv_object_broadcast : virtual public nerv_object {
   friend class _nerv_object_broadcast;
public:
   nerv_object_broadcast(msg_code_t _code);
   nerv_object_broadcast(msg_code_t _code, const nerv_object& __nerv_object);
   virtual ~nerv_object_broadcast(void);
   nerv_err_t broadcast(const void* _data_, pksize_t _size);
   nerv_err_t broadcast(ip_t _ip, const void* _data_, pksize_t _size);
private:
   _nerv_object_broadcast* _mp;
};
} // namespace Nerv35
```

☞ Object의 Broadcast 트랜잭션을 추상화한다.

☞ Broadcast 트랜잭션 타입의 메서드 클래스는 이 클래스를 일반 상속받아 구현한다.

- 상속받을 때 가상 상속(virtual)받으면 안 된다.
- 가상 상속 받으면 Broadcast Method는 오직하나만 존재할 수밖에 없기 때문이다.

2.3.7.1. 생성자 & 소멸자

2.3.7.1.1. nerv_object_broadcast(msg_code_t _code)

메서드	nerv_object_broadcast		
설명	_code로 식별되는 Broadcast 트랜잭션을 생성합니다.		
파라미터	타입	이름	설명
	msg_code_t	_code	Broadcast 식별 코드
Remark	▶ 여기에서 _code는 Broadcast의 의미에 대한 식별 코드값이다. 예를 들면 code=0x0001은 name broadcast, code=0x0002는 time broadcast 정보와 같이 구분하는데 사용 된다. ▶ 제약사항 ☞ _code = 0x0000은 사용할 수 없습니다.		

2.3.7.1.2. virtual ~nerv_object_broadcast()

메서드	~nerv_object_broadcast
설명	소멸자
Remark	☞ 가상(virtual) 소멸자이다.

2.3.7.2. 메서드

2.3.7.2.1. nerv_err_t broadcast(const void* _data_, pksize_t _size)

메서드	broadcast		
설명	주어진 데이터와 함께 같은 네트워크 안에 Broadcast 한다.		
파라미터	타입	이름	설명
	const void*	_data_	Broadcast 데이터 버퍼
	pksize_t	_size	Broadcast 데이터 크기
리턴	nerv_err_t		
	에러	발생원인 / 조치	
	NERV_ERR_NONE_	메서드 실행 성공	
	NERV_ERR_IS_NOT_CREATED_	객체를 create하지 않음 / 코드 로직 수정	
	NERV_ERR_EMPTY_NERV_MEMORY_	Nerv 메모리가 모자람 / Nerv 재시작	
Remark	▶ 같은 네트워크, 같은 도메인에 있는 모든 Proxy 객체에게 broadcast로 메시지를 전송한다. ▶ 이때, Proxy객체 또한 대응되는 nerv_proxy_object_broadcast를 작성하여야 수신할 수 있다.		

2.4. Proxy Object 클래스

2.4.1. nerv_proxy_object

☞ /Nerv35/include/Nerv35/nerv_proxy_object.h

```
namespace Nerv35 {
class NERV_SDK_DLL nerv_proxy_object {
   friend class Nerv_processor;
   friend class _nerv_proxy_object;
public:
   nerv_proxy_object();
   nerv_proxy_object(const nerv_proxy_object& __proxy_object);
   virtual ~nerv_proxy_object();
   nerv_err_t start(nerv_addr_t _stub_addr);
   nerv_err_t stop();
   bool       is_connected();
   bool       is_started();
   nerv_err_t get_address(nerv_addr_t* _proxy_addr_, nerv_addr_t* _stub_addr_);
protected:
   virtual void on_connect();
   virtual void on_disconnect();
protected:
   _nerv_proxy_object* _pp;
};
} // namespace Nerv35
```

☞ Object를 Proxy 추상화한다.

☞ Proxy Object Instance의 최상위 클래스이다.

☞ 상속받을 때 가상 상속(virtual)받아야 한다.

2.4.1.1. 생성자 & 소멸자

Nerv 에서는 생성자와 소멸자 이외에 별도의 start 함수와 stop 함수를 두어서 라이프 사이클에 대해서 이중적인 구조를 가진다. 이것은 C/C++ 언어의 기본적인 라이프 사이클 이외에 분산처리 환경에서 수시로 반복되는 라이프사이클의 비동기적 환경에 대응하기 위한 것이다. 따라서 실제로 Nerv의 C/C++ 관점의 생성자와 소멸자는 start/stop 과정을 위한 준비와 정리 동작만을 수행한다.

2.4.1.1.1. nerv_proxy_object()

메서드	nerv_proxy_object
설명	생성자
Remark	▶ 유일한 생성자

2.4.1.1.2. virtual ~nerv_object_call()

메서드	~nerv_object_call
설명	소멸자
Remark	▶ 가상(virtual) 소멸자이다.

2.4.1.2. 메서드

2.4.1.2.1. nerv_err_t start(nerv_addr_t _stub_addr)

메서드	start		
설명	in_domin 도메인의 in_server_ip : in_object_id 의 객체에 대한 Proxy 객체를 생성		
파라미터	타입	이름	설명
	nerv_addr_t	_stub_addr	stub의 IP, port 주소
리턴	nerv_err_t		
	에러	발생원인 / 조치	
	NERV_ERR_NONE_	메서드 실행 성공	
	NERV_ERR_ALREADY_CREATED__	이미 이 Instance로 start 함 / 로직 수정	
	NERV_ERR_EMPTY_NERV_MEMORY_	메모리가 모자람	
Remark	▶ stub의 ip와 port 주소로 stub에 연결을 시작한다.		

2.4.1.2.2. nerv_err_t stop()

메서드	stop	
설명	create한 객체를 더 이상 사용하지 않기 위해서 destroy 함	
리턴	nerv_err_t	
	에러	발생원인 / 조치
	NERV_ERR_NONE_	메서드 실행 성공
	NERV_ERR_IS_NOT_CREATED_	ceate 하지 않은 Instance를 destroy 함 / 로직 수정
Remark	▶ stop한 Instance의 method를 더 이상 쓸수 없음 ▶ 다시 쓰기 위해서는 start 하여야 함	

2.4.1.2.3. bool is_connected()

메서드	is_connected	
설명	Stub Object와 connect한 상태인지 확인한다.	
리턴	bool	
	값	설명
	true	connect 상태
	false	disconnect 상태

2.4.1.2.4. bool is_started()

메서드	is_started	
설명	proxy object가 start한 Instance 인지 검사함	
리턴	bool	
	값	설명
	true	start 한 Instance
	false	start하지 않았거나, stop한 Instance

2.4.1.2.5. nerv_err_t get_address(nerv_addr_t*, nerv_addr*)

메서드	get_address		
설명	연결상태일 때 proxy가 연결한 proxy와 stub의 연결주소를 반환한다.		
파라미터	타입	이름	설명
	nerv_addr_t*	_proxy_addr_	proxy 주소를 반환
	nerv_addr_t*	_stub_addr_	stub 주소를 반환
리턴	nerv_err_t		
	에러	발생원인 / 조치	
	NERV_ERR_NONE_	주소 획득 성공	
	NERV_ERR_IS_NOT_CREATED_	연결되지 않음	
Remark	▶		

2.4.1.3. 콜백 함수

2.4.1.3.1. virtual void on_connected()

함수	on_connect
설명	Stub과 연결되면 호출됨
Remark	▶

2.4.1.3.2. virtual void on_disconnect()

함수	on_disconnect
설명	Stub 과 연결이 종료되면 호출됨
Remark	▶ 다음과 같은 경우에 호출되어질 수 있다. ☞ Stub Object의 노드를 종료함(비정상종료 포함) ☞ Stub Object의 노드에서 더 이상 같은 도메인의 객체가 존재하지 않음 ☞ 네트워크 상태가 불안정하여 Hearbit 통신이 유지되지 아니하여 TCP세션이 종료됨

2.4.2. nerv_proxy_object_call

☞ /Nerv35/include/Nerv35/nerv_proxy_object_call.h

```
namespace Nerv35 {
class NERV_SDK_DLL nerv_proxy_object_call : public virtual nerv_proxy_object {
public:
   nerv_proxy_object_call(msg_code_t _code);
   nerv_proxy_object_call(msg_code_t               _code,
                          const nerv_proxy_object& __proxy_object);
   virtual ~nerv_proxy_object_call();
   const return_promise_t async_call(const void* _call_data_,
                                     pksize_t    _call_size,
                                     double      _limit_time = 1.0);
   nerv_err_t             async_return(return_promise_t return_promise,
                                       void*           _return_data_,
                                       pksize_t*       _size_,
                                       double*         _time_);
   void                   cancel(nerv_err_t _error);
private:
   _nerv_proxy_object_call* _mp;
};
} // namespace Nerv35
```

☞ Object의 Call 트랜잭션을 Proxy 추상화한다.

☞ Call 트랜잭션 타입의 Proxy 메서드 클래스는 이 클래스를 일반 상속받아 구현한다.

- 상속받을 때 가상 상속(virtual)받으면 안 된다.
- 가상 상속 받으면 Call Method는 오직하나만 존재할 수밖에 없기 때문이다.

2.4.2.1. 생성자 & 소멸자

2.4.2.1.1. nerv_proxy_object_call(msg_code_t _code)

메서드	nerv_proxy_object_call		
설명	_code로 식별되는 Proxy Call 트랜잭션을 생성합니다.		
파라미터	타입	이름	설명
	msg_code_t	_code	Call 식별 코드
Remark	▶ 여기에서 _code는 Call의 의미에 대한 식별 코드값이다. ▶ 여기에서 code는 Call의 의미에 대한 식별 코드값이다. 예를 들면 code=0x0001 은 printf 함수로, code=0x0002는 scanf 함수로 의미를 결정할 수 있습니다. ▶ 제약사항 ☞ _code = 0x0000은 사용할 수 없습니다.		

2.4.2.1.2. virtual ~nerv_proxy_object_call()

메서드	~nerv_proxy_object_call
설명	소멸자
Remark	☞ 가상(virtual) 소멸자이다.

2.4.2.2. 메서드

2.4.2.2.1. void cancel(nerv_err_t _error)

메서드	cancel		
설명	Call Method를 호출한 것을 인위적으로 취소한다.		
파라미터	타입	이름	설명
	nerv_err_t	_error	cancel 원인인 에러 사유
Remark	▶ 정상적으로 이루어진 Call을 Roll Back 할 수 있도록 취소가 가능하다. ▶ stub object의 Call에서 on_call_cancel()가 호출되도록 한다. ▶ 각 함수의 에러값은 0x00000000~0x7FFFFFFF 범위에서 정의 ▶ 에러코드의 값은 각 함수코드마다 그 의미가 다를 수 있음		

2.4.2.2.2. const return_promise_t async_call(void*, pksize_t, double)

메서드	async_call		
설명	Call Method를 호출합니다.		
파라미터	타입	이름	설명
	const void*	_call_data_	Input Data 버퍼
	pksize_t	_call_size	Input Data 바이트 크기
	double	_limit_time	제한 시간(초단위)
리턴	return_promise_t		리턴 결과 동기화 객체
Remark	▶ async_return과 짝을 이루어서 동작한다. ▶ async_call은 메시지를 전달하고 즉시 return_promise_t를 반환한다. ☞ 여기서 반환된 return_promise_t를 async_return을 통해서 비동기 리턴을 동기화 시킨다. ☞ 즉, async_call은 어떠한 대기상태도 가지지 아니한다. ▶ 제한시간을 초과 하는 경우 async_return을 통해서 에러를 통보한다. ▶ 에러나 예외 상황은 async_return을 통해서 확인한다.		

2.4.2.2.3. nerv_err_t async_return(return_promise_t, void*, pksize_t*, double*)

<table>
<tr><td>메서드</td><td colspan="3">async_return</td></tr>
<tr><td>설명</td><td colspan="3">Call Method의 리턴을 기다립니다.</td></tr>
<tr><td rowspan="5">파라미터</td><td>타입</td><td>이름</td><td>설명</td></tr>
<tr><td>return_promise_t</td><td>return_promise</td><td>async_call 의 return promise</td></tr>
<tr><td>void*</td><td>pout_return_data</td><td>Return Data를 읽어올 버퍼</td></tr>
<tr><td>pksize_t*</td><td>pout_size</td><td>Return Data 버퍼의 크기/
반환 데이터 크기</td></tr>
<tr><td>double*</td><td>_time_</td><td>Call 소요 시간 버퍼</td></tr>
<tr><td rowspan="10">리턴</td><td colspan="3">nerv_err_t</td></tr>
<tr><td>에러</td><td colspan="2">발생원인 / 조치</td></tr>
<tr><td>NERV_ERR_NONE_</td><td colspan="2">메서드 실행 성공</td></tr>
<tr><td>NERV_ERR_IS_NOT_START_</td><td colspan="2">start하지 않은 Instance를 stop함 / 로직수정</td></tr>
<tr><td>NERV_ERR_OBJECT_IS_NOT_EXIST_</td><td colspan="2">stub 와 연결되지 않음</td></tr>
<tr><td>NERV_ERR_EMPTY_NERV_MEMORY_</td><td colspan="2">Nerv 메모리가 모두 소진됨 / Nerv 재시작</td></tr>
<tr><td>NERV_ERR_LOCAL_TIMEOUT_</td><td colspan="2">Nerv Center가 Busy 상태임 / 노드 부하가 많음
async_call의 in_limit_time이 타임오버 됨</td></tr>
<tr><td>NERV_ERR_NULL_RETURN_PROMISE__</td><td colspan="2">return_promise에 Null 포인터 전달함 /
async_call에서 리턴한 return promise를 전달</td></tr>
<tr><td>NERV_ERR_BOUNDOUT_BUFFER__</td><td colspan="2">데이터가 존재하는 메서드 인데 버퍼가 없거나
실제 리턴 데이터의 크기보다 작다 / 로직 수정</td></tr>
<tr><td>NERV_ERR_REMOTE_TIMEOUT_</td><td colspan="2">stub object의 노드에서 패킷이 타임아웃 되었다.</td></tr>
<tr><td>Remark</td><td colspan="3">▶ async_return은 대기 상태를 가지는 함수로 다음의 경우 대기 시간을 가진다.
☞ 정상적으로 return을 수신하는 경우 또는 NERV_ERR_BOUNDOUT_BUFFER_의 경우 모든 전달 경로 및 Call의 처리시간만큼 대기한다.
☞ return을 정상적으로 수신 받지 못한 경우 async_call에서 지시한 in_limit_time 만큼 대기한다. (NERV_ERR_LOCAL_TIMEOUT_)
▶ 다른 에러의 경우 대부분 즉시 리턴한다.</td></tr>
</table>

2.4.3. nerv_proxy_object_signal

☞ /Nerv35/include/Nerv35/nerv_proxy_object_signal.h

```
namespace Nerv35 {
class NERV_SDK_DLL nerv_proxy_object_signal : virtual public nerv_proxy_object {
public:
   nerv_proxy_object_signal(msg_code_t _code);
   nerv_proxy_object_signal(msg_code_t               _code,
                            const nerv_proxy_object& __proxy_object);
   virtual ~nerv_proxy_object_signal(void);
   const return_promise_t async_signal(const void* _signal_data_,
                                       pksize_t    _call_size,
                                       double      _limit_time = 1.0);
   nerv_err_t async_return(return_promise_t return_object, double* _time_ = 0);
   nerv_err_t signal(const void*    _signal_data_,
                     pksize_t const _signal_size,
                     double         _limit_time = 1.0,
                     double*        _time_ = 0);
   void       cancel(nerv_err_t _error);
private:
   _nerv_proxy_object_signal* _mp;
};
} // namespace Nerv35
```

☞ Object의 Signal 트랜잭션을 Proxy 추상화한다.

☞ Signal 트랜잭션 타입의 Proxy 메서드 클래스는 이 클래스를 일반 상속받아 구현한다.

- 상속받을 때 가상 상속(virtual)받으면 안 된다.
- 가상 상속 받으면 Signal Method는 오직하나만 존재할 수밖에 없기 때문이다.

2.4.3.1. 생성자 & 소멸자

2.4.3.1.1. nerv_proxy_object_signal(msg_code_t _code)

메서드	nerv_proxy_object_signal		
설명	_code로 식별되는 Proxy Signal 트랜잭션을 생성합니다.		
파라미터	타입	이름	설명
	msg_code_t	_code	Signal 식별 코드
Remark	▶ 여기에서 _code는 Signal의 의미에 대한 식별 코드값이다. ▶ 여기서 code는 시그널의 의미에 대한 식별 코드 값이다. 예를 들면 code=0x0001은 키보드 타이핑 시그널, code=0x0002는 마우스 시그널과 같이 구분하는데 사용 된다. ▶ 제약사항 ☞ _code = 0x0000은 사용할 수 없습니다.		

2.4.3.1.2. virtual ~nerv_proxy_object_signal()

메서드	~nerv_proxy_object_signal
설명	소멸자
Remark	☞ 가상(virtual) 소멸자이다.

2.4.3.2. 메서드

2.4.3.2.1. nerv_err_t signal(const void*, pksize_t, double)

<table>
<tr><td>메서드</td><td colspan="3">signal</td></tr>
<tr><td>설명</td><td colspan="3">Signal Method를 호출합니다.</td></tr>
<tr><td rowspan="4">파라미터</td><td>타입</td><td>이름</td><td>설명</td></tr>
<tr><td>const void*</td><td>_signal_data_</td><td>Input Data 버퍼</td></tr>
<tr><td>pksize_t</td><td>_signal_size</td><td>Input Data 바이트 크기</td></tr>
<tr><td>double</td><td>_limit_time</td><td>제한 시간(초단위)</td></tr>
<tr><td rowspan="11">리턴</td><td colspan="3">nerv_err_t</td></tr>
<tr><td colspan="2">에러</td><td>발생원인 / 조치</td></tr>
<tr><td colspan="2">NERV_ERR_NONE_</td><td>메서드 실행 성공</td></tr>
<tr><td colspan="2">NERV_ERR_IS_NOT_CREATED_</td><td>ceate 하지 않은 Instance를 signal함 /
로직 수정</td></tr>
<tr><td colspan="2">NERV_ERR_OBJECT_IS_NOT_EXIST_</td><td>stub object가 start 되지 아니함</td></tr>
<tr><td colspan="2">NERV_ERR_EMPTY_NERV_MEMORY_</td><td>Nerv 메모리가 모두 소진됨 / Nerv 재시작</td></tr>
<tr><td colspan="2"></td><td>Nerv Center가 종료됨 / Nerv 재시작</td></tr>
<tr><td colspan="2">NERV_ERR_LOCAL_TIMEOUT_</td><td>Nerv Center가 Busy 상태임 / 노드 부하가 많음
async_call의 in_limit_time이 타임오버 됨</td></tr>
<tr><td colspan="2">NERV_ERR_NULL_RETURN_PROMISE_</td><td>return_promise에 Null 포인터 전달함 /
async_call에서 리턴한 return promise를 전달</td></tr>
<tr><td colspan="2">NERV_ERR_REMOTE_TIMEOUT_</td><td>stub object의 노드에서 패킷이 타임아웃
되었다.</td></tr>
<tr><td>Remark</td><td colspan="3">▶ _time_ 소요시간에 double 형 변수의 주소를 넘기면 signal을 성공적으로 완료하는데 소요된 시간을 리턴해 준다.
▶</td></tr>
</table>

2.4.3.2.2. nerv_err_t async_return(return_promise_t, void*, pksize_t*, double*)

<table>
<tr><td>메서드</td><td colspan="3">async_return</td></tr>
<tr><td>설명</td><td colspan="3">Call Method의 리턴을 기다립니다.</td></tr>
<tr><td rowspan="3">파라미터</td><td>타입</td><td>이름</td><td>설명</td></tr>
<tr><td>return_promise_t</td><td>_ret_promise</td><td>async_signal 의 return promise</td></tr>
<tr><td>double*</td><td>_time_</td><td>Call 소요 시간 버퍼</td></tr>
<tr><td rowspan="11">리턴</td><td colspan="3">nerv_err_t</td></tr>
<tr><td>에러</td><td colspan="2">발생원인 / 조치</td></tr>
<tr><td>NERV_ERR_NONE_</td><td colspan="2">메서드 실행 성공</td></tr>
<tr><td>NERV_ERR_IS_NOT_START_</td><td colspan="2">start하지 않은 Instance를 stop함 / 로직수정</td></tr>
<tr><td>NERV_ERR_OBJECT_IS_NOT_EXIST_</td><td colspan="2">stub 와 연결되지 않음</td></tr>
<tr><td>NERV_ERR_EMPTY_NERV_MEMORY_</td><td colspan="2">Nerv 메모리가 모두 소진됨 / Nerv 재시작</td></tr>
<tr><td>NERV_ERR_LOCAL_TIMEOUT_</td><td colspan="2">Nerv Center가 Busy 상태임 / 노드 부하가 많음
async_call의 in_limit_time이 타임오버 됨</td></tr>
<tr><td>NERV_ERR_NULL_RETURN_PROMISE__</td><td colspan="2">return_promise에 Null 포인터 전달함 /
async_call에서 리턴한 return promise를 전달</td></tr>
<tr><td>NERV_ERR_BOUNDOUT_BUFFER__</td><td colspan="2">데이터가 존재하는 메서드 인데 버퍼가 없거나
실제 리턴 데이터의 크기보다 작다 / 로직 수정</td></tr>
<tr><td>NERV_ERR_REMOTE_TIMEOUT_</td><td colspan="2">stub object의 노드에서 패킷이 타임아웃 되었다.</td></tr>
<tr><td>Remark</td><td colspan="3">▶ async_return은 대기 상태를 가지는 함수로 다음의 경우 대기 시간을 가진다.
☞ 정상적으로 return을 수신하는 경우 또는 NERV_ERR_BOUNDOUT_BUFFER_의 경우 모든 전달 경로 및 Call의 처리시간만큼 대기한다.
☞ return을 정상적으로 수신 받지 못한 경우 async_call에서 지시한 in_limit_time 만큼 대기한다. (NERV_ERR_LOCAL_TIMEOUT_)
▶ 다른 에러의 경우 대부분 즉시 리턴한다.</td></tr>
</table>

2.4.3.2.3. void cancel(nerv_err_t _error)

메서드	cancel		
설명	Signal Method를 호출한 것을 인위적으로 취소한다.		
파라미터	타입	이름	설명
	nerv_err_t	_error	cancel 원인인 에러 사유
Remark	▶ 정상적으로 이루어진 signal 처리를 Roll Back 할 수 있도록 취소가 가능하다. ▶ stub object의 signal에서 on_signal_cancel()가 호출되도록 한다. ▶ 각 함수의 에러값은 0x00000000~0x7FFFFFFF 범위에서 정의 ▶ 에러코드의 값은 각 Signal 코드마다 그 의미가 다를 수 있음		

2.4.4. nerv_proxy_object_event

☞ /Nerv35/include/Nerv35/nerv_proxy_object_event.h

```
namespace Nerv35 {
class NERV_SDK_DLL nerv_proxy_object_event : virtual public nerv_proxy_object {
   friend class _nerv_proxy_object_event;
public:
   nerv_proxy_object_event(msg_code_t _code);
   nerv_proxy_object_event(msg_code_t               _code,
                           const nerv_proxy_object& __proxy_object);
   virtual ~nerv_proxy_object_event();
   bool                   is_linked();
   const return_promise_t async_link(double _limit_time = 1.0);
   nerv_err_t async_link_return(return_promise_t ret_promise, double* _time_);
   void       unlink();
private:
   virtual void on_unlink();
   virtual void on_event(const void* _event_data_,
                         pksize_t    _event_size,
                         double      _elapsed_time);
private:
   _nerv_proxy_object_event* _mp;
};
} // namespace Nerv35
```

☞ Object의 Event 트랜잭션을 Proxy 추상화한다.

☞ Event 트랜잭션 타입의 Proxy 메서드 클래스는 이 클래스를 일반 상속받아 구현한다.

- 상속받을 때 가상 상속(virtual)받으면 안 된다.
- 가상 상속 받으면 Event Method는 오직하나만 존재할 수밖에 없기 때문이다.

2.4.4.1. 생성자 & 소멸자

2.4.4.1.1. nerv_proxy_object_event(msg_code_t _code)

메서드	nerv_proxy_object_event		
설명	_code로 식별되는 Proxy Event 트랜잭션을 생성합니다.		
파라미터	타입	이름	설명
	msg_code_t	_code	Event 식별 코드
Remark	▶ 여기에서 _code는 Event의 의미에 대한 식별 코드값이다. 예를 들면 code=0x0001은 키보드 타이핑 이벤트, code=0x0002는 마우스 이벤트과 같이 구분하는데 사용 된다. ▶ 제약사항 ☞ _code = 0x0000은 사용할 수 없습니다.		

2.4.4.1.2. virtual ~nerv_proxy_object_event()

메서드	~nerv_proxy_object_event
설명	소멸자
Remark	☞ 가상(virtual) 소멸자이다.

2.4.4.2. 메서드

2.4.4.2.1. bool is_linked()

메서드	is_linked
설명	Event 메서드가 Stub Object의 대응 Event에 연결 여부를 리턴한다.
리턴	bool
Remark	▶ 연결상태이면 true , 연결상태가 아니면 false

2.4.4.2.2. void unlink()

메서드	unlink
설명	Stub Object의 대응 Event와 연결상태를 종료한다.
Remark	▶ Stub Object의 대응 Event에서는 on_unlink이 호출된다.

2.4.4.2.3. const return_promise_t async_link(double in_limit_time = 1.0)

메서드	async_link		
설명	Stub Object 의 대응 Event에 연결하도록 비동기 요청합니다.		
파라미터	타입	이름	설명
	double	in_limit_time	제한 시간(초단위)
리턴	return_promise_t		리턴 결과 동기화 객체
Remark	▶ async_link_return과 짝을 이루어서 동작한다. ▶ async_link는 connect 메시지를 전달하고 즉시 return_promise를 반환한다. ☞ 여기서 반환된 return_promise를 async_link_return을 통해서 비동기 리턴을 동기화 시킨다. ☞ 즉, async_link는 어떠한 대기상태도 가지지 아니한다. ▶ 제한시간을 초과 하는 경우 async_link_return을 통해서 에러를 통보한다. ▶ 에러나 예외 상황은 async_link_return을 통해서 확인한다. ▶ link는 다른 트랜잭션의 link과정과 동일하며, Call의 방식과 유사하다.		

2.4.4.2.4. nerv_err_t async_link_return(return_promise_t, double*)

☞ nerv_err_t async_link_return(return_promise_t return_promise, double* _time_)

<table>
<tr><td>메서드</td><td colspan="3">async_link_return</td></tr>
<tr><td>설명</td><td colspan="3">Event Connect 요청의 리턴을 기다립니다.</td></tr>
<tr><td rowspan="3">파라미터</td><td>타입</td><td>이름</td><td>설명</td></tr>
<tr><td>return_promise_t</td><td>return_promise</td><td>async_link 의 return promise</td></tr>
<tr><td>double*</td><td>_time_</td><td>link 소요 시간 버퍼</td></tr>
<tr><td rowspan="10">리턴</td><td colspan="3">nerv_err_t</td></tr>
<tr><td colspan="2">에러</td><td>발생원인 / 조치</td></tr>
<tr><td colspan="2">NERV_ERR_NONE_</td><td>메서드 실행 성공</td></tr>
<tr><td colspan="2">NERV_ERR_IS_NOT_CREATED_</td><td>ceate하지 않은 Instance를 destroy함 / 로직수정</td></tr>
<tr><td colspan="2">NERV_ERR_OBJECT_IS_NOT_EXIST_</td><td>stub object가 start 되지 아니함</td></tr>
<tr><td colspan="2">NERV_ERR_EMPTY_NERV_MEMORY_</td><td>Nerv 메모리가 모두 소진됨 / Nerv 재시작</td></tr>
<tr><td colspan="2">NERV_ERR_LOCAL_TIMEOUT_</td><td>Nerv Center가 Busy 상태임 / 노드 부하가 많음
async_call의 in_limit_time이 타임오버 됨</td></tr>
<tr><td colspan="2">NERV_ERR_NULL_RETURN_PROMISE_</td><td>return_promise에 Null 포인터 전달함 / async_call에서 리턴한 return promise를 전달</td></tr>
<tr><td colspan="2">NERV_ERR_REMOTE_TIMEOUT_</td><td>stub object의 노드에서 패킷이 타임아웃 되었다.</td></tr>
<tr><td>Remark</td><td colspan="3">▶ async_link_return은 대기 상태를 가지는 함수로 다음의 경우 대기 시간을 가진다.
☞ 정상적으로 return을 수신하는 경우 모든 전달 경로 및 Stub Object의 Event on_link의 처리시간만큼 대기한다.
☞ return을 정상적으로 수신 받지 못한 경우 async_link 에서 지시한 in_limit_time 만큼 대기한다. (NERV_ERR_LOCAL_TIMEOUT_)
▶ 다른 에러의 경우 대부분 즉시 리턴한다.</td></tr>
</table>

2.4.4.3. 콜백 함수

2.4.4.3.1. virtual void on_unlink()

함수	on_unlink
설명	Event 연결이 Stub Object로부터 종료됨
Remark	▶ 순수 가상함수이므로 인스턴스화 하기 전에 반드시 구현하여야 함 ▶ unlink 동작을 구현하지 않더라도 오버라이딩하여 구현내용이 없음을 명시함 ▶ Stub Object의 대응하는 Event 와 link 상태에 있어야 on_unlink가 호출되어짐 ▶ 다음과 같은 원인으로 on_unlink 가 호출되어질 수 있음 ☞ Stub Object에서 unlink 를 호출 ☞ Stub Objecr가 stop됨 ☞ Stub Object의 process 가 종료됨(비정상 종료 포함) ☞ 네트워크 불안정으로 TCP 세션이 종료됨 ▪ hearbit 통신이 일정시간 이상 교류되지 아니함 ☞ Stub Object의 노드를 종료함(비정상 종료 포함)

2.4.4.3.2. virtual void on_evnet(const void*, pksize_t, double)

함수	on_event		
설명	Stub Object로부터 발생한 Event를 수신함		
파라미터	타입	이름	설명
	const void*	_event_data_	동기화 데이터
	pksize_t	_event_size	동기화 데이터 크기
	double	_elapsed_time	동기화 진행 시간
Remark	▶ 순수 가상함수이므로 인스턴스화 하기 전에 반드시 구현하여야 함 ▶ link 상태이어야 on_event가 수신됨 ☞ async_link/async_link_return 참조		

2.4.5. nerv_proxy_object_information

☞ /Nerv35/include/Nerv35/nerv_proxy_object_information.h

```
namespace Nerv35 {
class NERV_SDK_DLL nerv_proxy_object_information
    : virtual public nerv_proxy_object {
  friend class _nerv_proxy_object_information;
public:
  nerv_proxy_object_information(msg_code_t _code);
  nerv_proxy_object_information(msg_code_t              _code,
                                const nerv_proxy_object& __proxy_object);
  virtual ~nerv_proxy_object_information(void);
  bool                   is_linked();
  const return_promise_t async_link(bool   _reliability = false,
                                    double _limit_time = 1.0);
  nerv_err_t async_link_return(return_promise_t ret_promise, double* _time_);
  void       unlink();
  void       set_sync(bool _use_on_sync);
  letter_case_t see_history(size_t _size);
  void         set_history_size(size_t _history_size);
  nerv_err_t    read(void* _data_, pksize_t* _size_);
private:
  virtual void on_unlink();
  virtual void on_sync(Sync_model  _sync_model,
                       double      _elapsed_time,
                       double      _limit,
                       const void* _buffer_,
                       pksize_t    _size);
private:
  _nerv_proxy_object_information* _mp;
};
} // namespace Nerv35
```

☞ Object의 Information 트랜잭션을 Proxy 추상화한다.

☞ Information 트랜잭션 타입의 Proxy 메서드 클래스는 이 클래스를 일반 상속받아 구현한다.

- 상속받을 때 가상 상속(virtual)받으면 안 된다.
- 가상 상속 받으면 Information Method는 오직하나만 존재할 수밖에 없기 때문이다.

2.4.5.1. 생성자 & 소멸자

2.4.5.1.1. nerv_proxy_object_information(msg_code_t _code)

메서드	nerv_proxy_object_information		
설명	_code로 식별되는 Proxy Information 트랜잭션을 생성합니다.		
파라미터	타입	이름	설명
	msg_code_t	_code	Information 식별 코드
Remark	▶ 여기에서 _code는 Information의 의미에 대한 식별 코드값이다. 예를 들면 code=0x0001은 Sound 정보, code=0x0002는 Image 정보와 같이 구분하는데 사용 된다. ▶ 제약사항 ☞ _code = 0x0000은 사용할 수 없습니다.		

2.4.5.1.2. virtual ~nerv_proxy_object_information()

메서드	~nerv_proxy_object_information
설명	소멸자
Remark	☞ 가상(virtual) 소멸자이다.

2.4.5.2. 메서드

2.4.5.2.1. bool is_linked()

메서드	is_linked	
설명	Information 메서드가 Stub Object의 대응 Information에 연결 여부를 리턴한다.	
리턴	bool	연결상태이면 true , 연결상태가 아니면 false

2.4.5.2.2. void unlink()

메서드	unlink
설명	Stub Object의 대응 Information와 연결상태를 종료한다.
Remark	▶ Stub Object의 대응 Information에서는 on_unlink이 호출된다.

2.4.5.2.3. const return_promise_t async_link(bool , double)

메서드	async_link		
설명	Stub Object 의 대응 Information에 연결하도록 비동기 요청합니다.		
파라미터	타입	이름	설명
	bool	_reliability	신뢰성 요구(true)
	double	_limit_time	제한 시간(초단위)
리턴	return_promise_t		리턴 결과 동기화 객체
Remark	▶ in_udp_request에 따라 Stub Object는 동기 데이터를 전송한다. ▶ async_link_return과 짝을 이루어서 동작한다. ▶ async_link는 connect 메시지를 전달하고 즉시 return_promise를 반환한다. ☞ 여기서 반환된 return_promise를 async_link_return을 통해서 비동기 리턴을 동기화 시킨다. ☞ 즉, async_link는 어떠한 대기상태도 가지지 아니한다. ▶ 제한시간을 초과 하는 경우 async_link_return을 통해서 에러를 통보한다. ▶ 에러나 예외 상황은 async_link_return을 통해서 확인한다. ▶ link는 다른 트랜잭션의 link과정과 동일하며, Call의 방식과 유사하다.		

2.4.5.2.4. nerv_err_t async_link_return(return_promise_t, double*)

<table>
<tr><td>메서드</td><td colspan="3">async_link_return</td></tr>
<tr><td>설명</td><td colspan="3">Information Connect 요청의 리턴을 기다립니다.</td></tr>
<tr><td rowspan="3">파라미터</td><td>타입</td><td>이름</td><td>설명</td></tr>
<tr><td>return_promise_t</td><td>return_promise</td><td>async_link 의 return promise</td></tr>
<tr><td>double*</td><td>_time_</td><td>connect 소요 시간 버퍼</td></tr>
<tr><td rowspan="10">리턴</td><td colspan="3">nerv_err_t</td></tr>
<tr><td colspan="2">에러</td><td>발생원인 / 조치</td></tr>
<tr><td colspan="2">NERV_ERR_NONE_</td><td>메서드 실행 성공</td></tr>
<tr><td colspan="2">NERV_ERR_IS_NOT_CREATED_</td><td>ceate하지 않은 Instance를 destroy함 /
로직수정</td></tr>
<tr><td colspan="2">NERV_ERR_OBJECT_IS_NOT_EXIST_</td><td>stub object가 start 되지 아니함</td></tr>
<tr><td colspan="2">NERV_ERR_EMPTY_NERV_MEMORY_</td><td>Nerv 메모리가 모두 소진됨 / Nerv 재시작</td></tr>
<tr><td colspan="2">NERV_ERR_LOCAL_TIMEOUT_</td><td>Nerv Center가 Busy 상태임 / 노드 부하가 많음
async_call의 in_limit_time이 타임오버 됨</td></tr>
<tr><td colspan="2">NERV_ERR_NULL_RETURN_PROMISE_</td><td>return_promise에 Null 포인터 전달함 /
async_call에서 리턴한 return promise를 전달</td></tr>
<tr><td colspan="2">NERV_ERR_REMOTE_TIMEOUT_</td><td>stub object의 노드에서 패킷이 타임아웃
되었다.</td></tr>
<tr><td>Remark</td><td colspan="3">▶ async_link_return은 대기 상태를 가지는 함수로 다음의 경우 대기 시간을 가진다.
☞ 정상적으로 return을 수신하는 경우 모든 전달 경로 및 Stub Object의 Information on_link의 처리시간만큼 대기한다.
☞ return을 정상적으로 수신 받지 못한 경우 async_link 에서 지시한 in_limit_time 만큼 대기한다. (NERV_ERR_LOCAL_TIMEOUT_)
▶ 다른 에러의 경우 대부분 즉시 리턴한다.</td></tr>
</table>

2.4.5.2.5. void set_sync(bool _use_on_sync)

메서드	set_sync		
설명	on_sync 함수를 호출해 줄지 지시한다.		
파라미터	타입	이름	설명
	bool	_use_on_sync	호출 true / 비호출 false
Remark	▶ 지시하지 않는 경우 기본적으로 호출 true 이다.		

2.4.5.2.6. letter_case_t see_history(size_t _size)

메서드	see_history		
설명	동기화 된 데이터의 History 기록을 가져온다.		
파라미터	타입	이름	설명
	size_t	_size	가져올 데이터의 최대 개수
리턴	letter_case_t		DLL history의 포인터
Remark	▶ 리턴 결과를 template_history의 생성자에 넣어서 history를 사용 할 수 있다. ▶ 배열형으로써 첨자 [] 방식으로 데이터를 접근할 수 있다. ▶ [n] 데이터가 [n+1]보다 최근이고, [0]은 가장 최근 데이터이다. ▶ set_history_size()를 통해서 최대 보관 크기를 지시할 수 있다. ▶ start 하지 않았거나 connect 되지 않은 경우 NULL 포인터를 리턴한다.		

2.4.5.2.7. void set_history_size(size_t _history_size)

메서드	set_history_size		
설명	History의 최대 보관 개수를 지시한다.		
파라미터	타입	이름	설명
	size_t	_history_size	보관할 데이터의 최대 개수

2.4.5.2.8. nerv_err_t read(void* _data_ , pksize_t* p_size)

메서드	read		
설명	가장 최근에 동기화 된 데이터를 읽어온다.		
파라미터	타입	이름	설명
	void*	_data_	읽어올 데이터 버퍼
	pksize_t*	_size_	읽어올 데이터 크기
리턴	nerv_err_t		
	에러	발생원인 / 조치	
	NERV_ERR_NONE_	메서드 실행 성공	
	NERV_ERR_IS_NOT_CREATED_	객체를 create하지 않음	
	NERV_ERR_EMPTY_HISTORY	connect 되지 않았거나, 동기화된 데이터가 없음	

2.4.5.3. 콜백 함수

2.4.5.3.1. virtual void on_unlink()

함수	on_unlink
설명	Information 연결이 Stub Object로부터 종료됨
Remark	▶ Stub Object의 대응하는 Information과 connect 상태에 있어야 on_disconnect가 호출되어짐 ▶ 다음과 같은 원인으로 on_disconnected 가 호출되어질 수 있음 ☞ Stub Object에서 disconnect 를 호출 ☞ Stub Objecr가 destroy됨 ☞ Stub Object의 process 가 종료됨(비정상 종료 포함) ☞ 네트워크 불안정으로 TCP 세션이 종료됨 ▪ hearbit 통신이 일정시간 이상 교류되지 아니함 ☞ Stub Object의 노드를 종료함(비정상 종료 포함)

2.4.5.3.2. virtual void on_sync(...)

함수	on_sync		
설명	Stub Object로 Information Data가 동기화 직후 호출됨		
파라미터	타입	이름	설명
	Sync_model	_sync_model	동기화 모델
	double	_elapsed_time	동기화 진행 시간
	double	in_limit	동기화 제한 시간
	const void*	_buffer_	동기화 데이터
	const pksize_t	_size	동기화 데이터 크기
Remark	▶ 동기화 직후(read 함수로 읽을 수 있는 상태)에 호출됨 ▶ 동기화 데이터를 이 함수 외부에서 읽기 위해서는 read 함수를 사용 ▶ 모든 동기화 데이터를 개별적으로 식별하고, 정확한 동기화 타이밍에 어떤 작업을 하기위해서는 이 함수에서 구현해야 한다.		

2.4.6. nerv_proxy_object_command

☞ /Nerv35/include/Nerv35/nerv_proxy_object_command.h

```
namespace Nerv35 {
class NERV_SDK_DLL nerv_proxy_object_command : virtual public nerv_proxy_object {
   friend class _nerv_proxy_object_command;
   friend class _nerv_proxy_object;
public:
   nerv_proxy_object_command(msg_code_t _code);
   nerv_proxy_object_command(msg_code_t               _code,
                             const nerv_proxy_object& __proxy_object);
   virtual ~nerv_proxy_object_command();
   bool                   is_linked();
   const return_promise_t async_link(double _limit_time = 1.0);
   nerv_err_t async_link_return(return_promise_t ret_promise, double* _time_);
   void       unlink();
   void       set_sync_model(bool       _reliability,
                             Sync_model _sync_model,
                             double     _interval);
   void*      get(pksize_t _size);
   void       cancel();
   nerv_err_t post(pksize_t _size);
   nerv_err_t write(const void* _data_, pksize_t _size);
   letter_case_t see_history(size_t _size);
   void          set_history_size(size_t _history_size);
   nerv_err_t    read(void* _data_, pksize_t* _size_);
protected:
   virtual void on_unlink();
protected:
   _nerv_proxy_object_command* _mp;
};
} // namespace Nerv35
```

☞ Object의 Command 트랜잭션을 Proxy 추상화한다.

☞ Command 트랜잭션 타입의 Proxy 메서드 클래스는 이 클래스를 일반 상속받아 구현한다.

- 상속받을 때 가상 상속(virtual)받으면 안 된다.
- 가상 상속 받으면 Command Method는 오직하나만 존재할 수밖에 없기 때문이다.

2.4.6.1. 생성자 & 소멸자

2.4.6.1.1. nerv_proxy_object_command(msg_code_t _code)

메서드	nerv_proxy_object_command		
설명	_code로 식별되는 Proxy Command 트랜잭션을 생성합니다.		
파라미터	타입	이름	설명
	msg_code_t	_code	Command 식별 코드
Remark	▶ 여기서 code는 Command의 의미에 대한 식별 코드 값이다. 예를 들면 code=0x0001은 조이스틱 핸들 명령, code=0x0002는 조이스틱 버튼 명령와 같이 구분하는데 사용 된다. ▶ 제약사항 ☞ _code = 0x0000은 사용할 수 없습니다.		

2.4.6.1.2. virtual ~nerv_proxy_object_command()

메서드	~nerv_proxy_object_command
설명	소멸자
Remark	☞ 가상(virtual) 소멸자이다.

2.4.6.2. 메서드

2.4.6.2.1. bool is_linked()

메서드	is_linked	
설명	Command 메서드가 Stub Object의 대응 Command에 연결 여부를 리턴한다.	
리턴	bool	연결상태이면 true , 연결상태가 아니면 false

2.4.6.2.2. void unlink()

메서드	unlink
설명	Stub Object의 대응 Command와 연결상태를 종료한다.
Remark	▶ Stub Object의 대응 Command에서는 on_disconnect이 호출된다.

2.4.6.2.3. void set_sync_model(bool, Sync_model, double)

메서드	set_sync_model		
설명	Command의 데이터 동기 방식을 선택합니다.		
파라미터	타입	이름	설명
	bool	_reliability	신뢰성
	Sync_model	_sync_model	동기화 방식
	double _interval	_interval	시간 간격
Remark	▶ _sync_model은 다음 두 개의 값중 하나 ☞ Sync_model::on_post ▪ post / write 함수를 호출하는 타이밍에 데이터를 동기화 ▪ 이때 _interval 은 패킷 제한 시간 ☞ Sync_model::on_timer ▪ Nerv 내부 타이머에 의하여 데이터를 주기적으로 동기화 ▪ 이때 _interval은 동기화 시간 간격		

2.4.6.2.4. const return_promise_t async_link(double)

메서드	async_link		
설명	Stub Object 의 대응 Command에 연결하도록 비동기 요청합니다.		
파라미터	타입	이름	설명
	double	in_limit_time	제한 시간(초단위)
리턴	return_promise_t		리턴 결과 동기화 객체
Remark	▶ async_link_return과 짝을 이루어서 동작한다. ▶ async_link는 link 메시지를 전달하고 즉시 return_promise를 반환한다. ☞ 여기서 반환된 return_promise를 async_link_return을 통해서 비동기 리턴을 동기화 시킨다. ☞ 즉, async_link는 어떠한 대기상태도 가지지 아니한다. ▶ 제한시간을 초과 하는 경우 async_link_return을 통해서 에러를 통보한다. ▶ 에러나 예외 상황은 async_link_return을 통해서 확인한다. ▶ link는 다른 트랜잭션의 link과정과 동일하며, Call의 방식과 유사하다.		

2.4.6.2.5. nerv_err_t async_link_return(return_promise_t, double*)

<table>
<tr><td>메서드</td><td colspan="3">async_link_return</td></tr>
<tr><td>설명</td><td colspan="3">Command link 요청의 리턴을 기다립니다.</td></tr>
<tr><td rowspan="3">파라미터</td><td>타입</td><td>이름</td><td>설명</td></tr>
<tr><td>return_promise_t</td><td>return_promise</td><td>async_link 의 return promise</td></tr>
<tr><td>double*</td><td>_time_</td><td>connect 소요 시간 버퍼</td></tr>
<tr><td rowspan="9">리턴</td><td colspan="3">nerv_err_t</td></tr>
<tr><td colspan="2">에러</td><td>발생원인 / 조치</td></tr>
<tr><td colspan="2">NERV_ERR_NONE_</td><td>메서드 실행 성공</td></tr>
<tr><td colspan="2">NERV_ERR_IS_NOT_CREATED_</td><td>ceate하지 않은 Instance를 destroy함 / 로직수정</td></tr>
<tr><td colspan="2">NERV_ERR_OBJECT_IS_NOT_EXIST_</td><td>stub object가 start 되지 아니함</td></tr>
<tr><td colspan="2">NERV_ERR_EMPTY_NERV_MEMORY_</td><td>Nerv 메모리가 모두 소진됨 / Nerv 재시작</td></tr>
<tr><td colspan="2">NERV_ERR_LOCAL_TIMEOUT_</td><td>Nerv Center가 Busy 상태임 / 노드 부하가 많음
async_call의 in_limit_time이 타임오버 됨</td></tr>
<tr><td colspan="2">NERV_ERR_NULL_RETURN_PROMISE_</td><td>return_promise에 Null 포인터 전달함 / async_call에서 리턴한 return promise를 전달</td></tr>
<tr><td colspan="2">NERV_ERR_REMOTE_TIMEOUT_</td><td>stub object의 노드에서 패킷이 타임아웃 되었다.</td></tr>
<tr><td>Remark</td><td colspan="3">▶ async_link_return은 대기 상태를 가지는 함수로 다음의 경우 대기 시간을 가진다.
☞ 정상적으로 return을 수신하는 경우 모든 전달 경로 및 Stub Object의 Command on_link의 처리시간만큼 대기한다.
☞ return을 정상적으로 수신 받지 못한 경우 async_link 에서 지시한 in_limit_time 만큼 대기한다. (NERV_ERR_LOCAL_TIMEOUT_)
▶ 다른 에러의 경우 대부분 즉시 리턴한다.</td></tr>
</table>

2.4.6.2.6. void* get(pksize_t _size)

메서드	get		
설명	동기화 할 데이터를 작성할 메모리를 가져온다.		
파라미터	타입	이름	설명
	pksize_t	_size	가져올 메모리의 크기
리턴	void*		메모리 주소
Remark	▶ information이나 command의 데이터 작성 함수 ☞ get/post(cancel) 방식 ▪ get 이후에는 반드시 cancel 로 취소하거나 post로 작성을 완료하여야 한다. ▪ Nerv의 메모리를 직접 획득하므로 데이터 카피가 발생하지 않는다. ☞ write 방식 ▪ get/post(cancel)방식은 아무래도 3개의 함수로 작성하기 때문에 코드가 복잡해 보일수 있어서 데이터의 크기 때문에 문제가 없는 것은 가독성을 위해서 write함수를 사용한다. ▪ 내부적으로 get/post(cancel) 과 memcpy 함수를 이용한다. ▪ 데이터 크기가 크면 유저 메모리에서 Nerv메모리로 복사하는 과정을 가지므로 속도 저하의 원인이 될 수도 있다. ▶ Nerv 메모리가 모자라는 경우 NULL 을 리턴한다. ▪ 0을 리턴한 경우 post/cancel 하지 않는다.		

2.4.6.2.7. void cancel()

메서드	cancel
설명	get으로 가져온 메모리를 취소한다.
Remark	☞ void* get(pksize_t _size) 참조

2.4.6.2.8. nerv_err_t post(pksize_t _size)

메서드	post		
설명	get으로 가져온 메모리에 데이터 작성을 완료한다.		
파라미터	타입	이름	설명
	pksize_t	_size	작성 완료한 데이터의 크기
리턴	nerv_err_t		
	에러	발생원인 / 조치	
	NERV_ERR_NONE_	메서드 실행 성공	
	NERV_ERR_IS_NOT_CREATED_	객체를 create하지 않음	
	NERV_ERR_DONT_GET_	get으로 메모리를 확보하지 않음	
	NERV_ERR_TOO_BIG_SIZE_	get으로 확보한 메모리보다 큰 크기로 post함	
Remark	☞ void* get(pksize_t _size) 참조		

2.4.6.2.9. nerv_err_t write(const void* _data_, pksize_t _size)

메서드	post		
설명	get으로 가져온 메모리에 데이터 작성을 완료한다.		
파라미터	타입	이름	설명
	const void*	_data_	작성할 원본 데이터 버퍼
	pksize_t	_size	작성할 원본 데이터 크기
리턴	nerv_err_t		
	에러	발생원인 / 조치	
	NERV_ERR_NONE_	메서드 실행 성공	
	NERV_ERR_IS_NOT_CREATED_	객체를 create하지 않음	
	NERV_ERR_EMPTY_NERV_MEMORY_	Nerv 메모리가 모자람	
Remark	▶ void* get(pksize_t _size) 참조		

2.4.6.2.10. letter_case_t see_history(size_t _size)

메서드	see_history		
설명	post/write 한 데이터의 History 기록을 가져온다.		
파라미터	타입	이름	설명
	size_t	_size	가져올 데이터의 최대 개수
리턴	letter_case_t		DLL history의 포인터
Remark	▶ 리턴 결과를 template_history의 생성자에 넣어서 history를 사용 할 수 있다. 배열형으로써 첨자 [] 방식으로 데이터를 접근할 수 있다. [n] 데이터가 [n+1]보다 최근이고, [0]은 가장 최근 데이터이다. ▶ set_history_size()를 통해서 최대 보관 크기를 지시할 수 있다. ▶ start 하지 않은 경우 NULL 포인터를 리턴한다.		

2.4.6.2.11. void set_history_size(size_t _history_size)

메서드	set_history_size		
설명	History의 최대 보관 개수를 지시한다.		
파라미터	타입	이름	설명
	size_t	_history_size	보관할 데이터의 최대 개수

2.4.6.2.12. nerv_err_t read(void* _data_, pksize_t* p_size)

메서드	read		
설명	가장 최근에 작성한 데이터를 읽어온다.		
파라미터	타입	이름	설명
	void*	_data_	읽어올 데이터 버퍼
	pksize_t*	_size_	읽어올 데이터 크기
리턴	nerv_err_t		
	에러		발생원인 / 조치
	NERV_ERR_NONE_		메서드 실행 성공
	NERV_ERR_IS_NOT_CREATED_		객체를 create하지 않음
	NERV_ERR_EMPTY_HISTORY		읽어올 데이터가 없음

2.4.6.3. 콜백 함수

2.4.6.3.1. virtual void on_unlink()

함수	on_unlink
설명	Command 연결이 Stub Object로부터 종료됨
Remark	▶ Stub Object의 대응하는 Command와 link 상태에 있어야 on_unlink가 호출되어짐 ▶ 다음과 같은 원인으로 on_disconnected 가 호출되어질 수 있음 ☞ Stub Object에서 disconnect 를 호출 ☞ Stub Objecr가 destroy됨 ☞ Stub Object의 process 가 종료됨(비정상 종료 포함) ☞ 네트워크 불안정으로 TCP 세션이 종료됨 ▪ hearbit 통신이 일정시간 이상 교류되지 아니함 ☞ Stub Object의 노드를 종료함(비정상 종료 포함)

2.4.7. nerv_proxy_object_broadcast

☞ /Nerv35/include/Nerv35/nerv_proxy_object_broadcast.h

```
namespace Nerv35 {
class NERV_SDK_DLL nerv_proxy_object_broadcast
    : virtual public nerv_proxy_object {
  friend class _nerv_proxy_object_broadcast;
public:
  nerv_proxy_object_broadcast(msg_code_t _code);
  nerv_proxy_object_broadcast(msg_code_t              _code,
                              const nerv_proxy_object& __proxy_object);
  virtual ~nerv_proxy_object_broadcast(void);
protected:
  virtual void on_broadcast(nerv_addr_t _stub_addr,
                            const void* _buf_,
                            pksize_t    _size,
                            double      _elapsed_time);
private:
  _nerv_proxy_object_broadcast* _mp;
};
} // namespace Nerv35
```

☞ Object의 Broadcast 트랜잭션을 Proxy 추상화한다.

☞ Broadcast 트랜잭션 타입의 Proxy 메서드 클래스는 이 클래스를 일반 상속받아 구현한다.

- 상속받을 때 가상 상속(virtual)받으면 안 된다.
- 가상 상속 받으면 Broadcast Method는 오직하나만 존재할 수밖에 없기 때문이다.

2.4.7.1. 생성자 & 소멸자

2.4.7.1.1. nerv_proxy_object_broadcast(msg_code_t _code)

메서드	nerv_proxy_object_broadcast		
설명	_code로 식별되는 Proxy Broadcast 트랜잭션을 생성합니다.		
파라미터	타입	이름	설명
	msg_code_t	_code	Broadcast 식별 코드
Remark	▶ 여기에서 _code는 Broadcast의 의미에 대한 식별 코드값이다. 예를 들면 code=0x0001은 name broadcast, code=0x0002는 time broadcast 정보와 같이 구분하는데 사용 된다. ▶ 제약사항 ☞ _code = 0x0000은 사용할 수 없습니다.		

2.4.7.1.2. virtual ~nerv_proxy_object_broadcast()

메서드	~nerv_proxy_object_broadcast
설명	소멸자
Remark	☞ 가상(virtual) 소멸자이다.

2.4.7.2. 콜백 함수

2.4.7.2.1. virtual void on_broadcast(nerv_addr_t, void*, pksize_t, double)

함수	on_broadcast		
설명	Stub Object로부터 발생한 Event를 수신함		
파라미터	타입	이름	설명
	nerv_addr_t	_stub_addr	Broadcast 생성 Stub 주소
	const void*	_buf_	Broadcast 데이터 버퍼
	pksize_t	_size	Broadcast 데이터 크기
	double	_elapsed_time	Broadcast 진행 시간
Remark	▶ 동일 네트워크 상에서 존재하는 Stub에서 전송한 데이터는 구별없이 수신된다.		

2.5. History

History는 Information이나 Command에서 post/write하거나 동기화된 데이터를 시간순서로 보기위한 방법이다. see_history 메서드를 통해서 letter_case_t 이라는 포인터 타입을 리턴받으면 이 포인터를 template_history 의 생성자에 넣으면 소멸자가 호출될 때까지 데이터를 유지하도록 설계되어 있다.

template_history를 사용하기에 앞서 하나의 엘리먼트를 나타내는 template_history_elem 템플릿 클래스와 Nerv SDK DLL의 일부분인 nerv_history를 먼저 살펴본 후에 template_history 템플릿 클래스를 설명할 것이다.

2.5.1. template_history_elem

☞ /Nerv35/include/Nerv35/nerv_proxy_object.h

```
namespace Nerv35 {
template <class TYPE> struct template_history_elem {
   TYPE*        buffer_;
   pksize_t     size;
   double       interval;
   nerv_clock_t clock;
};
}
```

▶ TYPE 형식의 데이터를 history에 사용할 때 배열구조의 history에서 하나의 엘리먼트를 표현할 때 나타내는 구조체이다. 즉, history는 template_history_elem의 배열 개념이다.

▶ 멤버

☞ p_buffer

- TYPE형식의 데이터를 보관하고있는 메모리 버퍼 주소

☞ size

- 메모리 버퍼 주소의 크기
- 보통 데이터의 크기는 약속되어져 있으나, 가변 크기의 데이터인 경우 size 정보가 필요하다.

☞ interval

- 바로 직전 데이터와의 시간 간격(초단위)

☞ clock

- 데이터를 post/write하거나 동기화 되었을때의 clock값(밀리초 단위)
- 시스템이 부팅할 때부터의 시간값

2.5.2. nerv_history

☞ /Nerv35/include/Nerv35/nerv_history.h

```
namespace Nerv35 {
typedef class NERV_SDK_DLL nerv_history {
public:
   nerv_history();
   nerv_history(const nerv_history& destructure);
   nerv_history(letter_case_t _letter_case);
   ~nerv_history();
                              operator bool();
   nerv_history&              operator=(letter_case_t _letter_case);
   nerv_history&              operator=(nerv_history& destructure);
   template_history_elem<void> operator[](unsigned int _index);
   size_t                size();
   void                       clear();
   bool                       empty();

private:
   letter_case_t letter_case;
} destructor_t;
}
```

void 타입의 엘리먼트 다루는 history 클래스이다. 타입을 void로 표현하기 때문에 이 클래스의 데이터는 어떠한 형식으로든 만들어질 수 있다. history의 순수한 기능만을 표현한 클래스이다.

▶ nerv_history()

☞ 비어있는 History를 만드는 생성자

▶ nerv_history(const nerv_history& destructure);

☞ 다른 History로부터 History를 만드는 생성자

▶ nerv_history(letter_case_t in_letter_case);

☞ see_history() 메서드의 리턴인 letter_case로부터 History를 만드는 생성자

▶ ~nerv_history();

☞ 소멸자

☞ Nerv 메모리를 Free시켜주는 역할을 수행한다.

▶ operator bool();

☞ history 객체가 조건문에 들어가는 경우 bool 형으로 자동 형변환하게 하며, History가 비어있는 경우 false, 내용물이 있으면 true를 반환한다.

☞ 엘리먼트가 없어도, letter_case나 다른 History로부터 생성된 것이면 true를 반환한다.

▶ nerv_history& operator = (letter_case_t in_letter_case);

☞ 할당문을 통해서 see_history() 메서드의 리턴인 letter_case로부터 History를 만들 수 있다.

▶ nerv_history& operator = (nerv_history& destructure);

☞ 할당문을 통해서 다른 history 객체에서 History를 만들 수 있다.

▶ template_history_elem<void> operator [] (unsigned int in_index);

☞ 배열 첨자를 이용하여 엘리먼트에 접근할 수 있다.

▶ size_t size();

☞ 엘리먼트의 개수를 반환한다.

▶ void clear();

☞ History 내용을 지운다.

▶ bool empty();

☞ 엘리먼트가 비어있으면 true, 엘리먼트가 있으면 false를 반환한다.

2.5.3. template_history

☞ /Nerv35/include/Nerv35/nerv_history.h

```
namespace Nerv35 {
template <class TYPE> class template_history : public nerv_history {
public:
   template_history() {}
   template_history(const template_history& destructure) : nerv_history(destructure) {}
   template_history(letter_case_t _letter_case) : nerv_history(_letter_case) {}
                operator bool() { return *(nerv_history*)(this); }
   template_history& operator=(letter_case_t _letter_case)
   {
     *((nerv_history*)(this)) = _letter_case;
     return (*this);
   }
   template_history& operator=(template_history& destructure)
   {
     *(nerv_history*)(this) = *(nerv_history*)&destructure;
     return (*this);
   }
   template_history_elem<TYPE> operator[](unsigned int _index)
   {
     template_history_elem<void> void_elem = (*((nerv_history*)(this)))[_index];
     template_history_elem<TYPE> elem;
     elem.buffer_ = (TYPE*)void_elem.buffer_;
     elem.size = void_elem.size;
     elem.clock = void_elem.clock;
     elem.interval = void_elem.interval;
     return elem;
   }
   size_t size() { return nerv_history::size(); }
   void        clear() { nerv_history::clear(); }
   bool        empty() { return nerv_history::empty(); }
};
} // namespace Nerv35
```

▶ nerv_history 로부터 상속받아서 아무 타입이나 History로 사용가능하게 끔 포장한 것이다.

▶ ex

☞ template_history<int> int_history;

- 정수형 데이터를 데이터로서 template_history_elem<int>를 엘리먼트로 갖는 history 객체를 선언한 것이다.

▶ 예제

```
typedef template_history<int>       int_history_t;
typedef template_history_elem<int>  int_history_elem_t;

if( int_history_t history = see_history() )
{
    size_t size = history.size();
    for(size_t i=0; i< size; i++)
    {
        history[i].p_buffer;
        history[i].size;
        history[i].interval;
        history[i].clock;
    }
}
```

☞ 일반적으로 위와 같이 간단하게 history 데이터를 접근하여 사용할수 있다.

2.6. 유틸리티 함수들

☞ /Nerv35/include/Nerv35/nerv_utility.h

```
namespace Nerv35 {
NERV_SDK_DLL size_t get_address(const char* _name_, nerv_addr_t* _stub_addr_);
NERV_SDK_DLL ip_t  inet_addr(const char* _ip_);
NERV_SDK_DLL char* inet_ntoa(ip_t _ip);
NERV_SDK_DLL nerv_clock_t get_cpu_clock();
NERV_SDK_DLL nerv_clock_t get_cpu_hz();
NERV_SDK_DLL void initialize();
NERV_SDK_DLL void initialize(size_t _thread_count,
                             size_t _call_buffer_count,
                             size_t _udp_buffer_count);
NERV_SDK_DLL void finalize();
} // namespace Nerv35
```

2.6.1. 초기화 준비 & 마지막 정리

2.6.1.1. void initialize(size_t ,size_t, size_t);

▶ nerv 클래스들을 사용하기 위해서는 반듯이 이 함수를 소프트웨어 실행 초반에 실행해야 한다.

▶ _thread_count는 Nerv 콜백함수들을 호출하는 스레드의 개수를 지시하는 것이다.

▶ _call_buffer_count 는 call 스타일의 메시지 처리시 버퍼링 하는 최대 개수이다.

▶ _udp_buffer_count 는 udp 메시지 처리시 버퍼링하는 최대 개수이다.

2.6.1.2. void finalize();

▶ 프로그램을 종료하기 전에 정리하는 함수이다.

▶ 혹시나 있을수 있는 Nerv의 메모리 누수를 막기위해 사용하고 종료하는 것이 좋다.

☞ 즉, 필수로 사용해야 하는 것은 아니다.

2.6.2. IP 주소 변환

<Winsock.h> 에 구현되어 있지만, winsock을 include해야 하고, ws32_2.lib 라이브러리를 링크해야 하는 귀찮음을 해결해주는 Nerv 네임스페이스에 정의된 inet_addr 함수와 inet_ntoa 함수이다.

2.6.2.1. ip_t inet_addr(const char_t* str_ip);

▶ 멀티바이트 문자열의 IP 주소를 ip_t 타입으로 변환하는 함수

2.6.2.2. char* inet_ntoa(const ip_t& ip);

▶ ip_t 타입의 IP 주소를 멀티바이트 문자열로 변환하는 함수

2.6.2.3. size_t get_address(const char* _name_, nerv_addr_t* _stub_addr_);

▶ Nmed Object 주소획득하는 함수이다.

▶ _name_을 지시하면 현재 로컬네트워크에 실행되고 있는 객체의 주소를 얻어올수 있다.

▶ 이를 통해서 proxy 객체를 생성하는 방법이 가능하다.

2.6.3. CPU 클럭

☞ Nerv 내부적으로 사용하는 clock 함수를 응용 개발자가 직접 사용할수 있도록 오픈한 것이다.

☞ history나 기타 등등에서 볼수 있는 clock_t 형식의 값들은 아래 함수에 의하여 계산된 것이다.

2.6.3.1. nerv_clock_t get_cpu_clock();

☞ cpu의 클럭을 리턴한다.

2.6.3.2. nerv_clock_t get_cpu_hz();

☞ cpu의 동작 주파수를 리턴한다.

3. Nerv Programing

3.1. Multi Thread Programing

3.1.1. Nerv DLL 초기화

▶ Nerv::initialize(size_t thread_count = 1)

☞ Nerv의 Dll을 초기화 시키면 Call Back Function를 호출할 수 있도록 하는 Thread를 생성시킨다.

☞ 이때 기본적으로 1개를 생성하고, 인자를 지시하여 여러 개의 Thread가 Call Back Function을 호출할 수 있도록 해준다.

☞ 추가적으로 Nerv를 관리하는 Thread 1개가 생성되지만 이 Thread는 Application Source Code에 관여하지 않는 독립 Thread이다.

☞ 즉, initialize 함수는 1개의 독립 Thread와 1개 이상의 Call Back Thread를 생성한다.

3.1.2. Call Back Function = on_ virtual function

▶ Nerv의 Call Back 함수는 C++ 언어로 virtual function으로 작성되어 있고, function의 이름이 접두사로 on_를 붙여서 명명되어져 있어서 on_ virtual function 이라고도 부른다.

▶ on_ virtual function은 Application Process의 Main Thread 와는 다른 Thread에서 동작한다.

▶ 때문에 Nerv를 이용하여 프로그램 하는 것은 Multi Thread Programing을 하는 것이다.

3.1.3. Multi Threading 문제점 인식

▶ Multi Threading을 하는 경우 다음과 같은 문제점에 놓일 수 있다는 것에 유의하여서 Programing 해야 한다.

☞ Race Condition

- 동일한 Resource에 대하여 여러 Thread가 동시에 접근하여 Thread간 실행 순서 때문에 결과가 달라지는 현상이다.
- 이를 해결하기 위해 Mutual Exclusion(Critical Section, Mutex) 나 Conditional Synchronization (Semaphore , Event) 과 같은 동기화 객체를 사용한다.

☞ Dead Lock

- Race Condition을 해결해기 위해 동기화 객체를 사용했을 때 서로 다른 Thread가 동시에 각각 Resource를 선점함과 동시에 다른 Thread가 가진 Resource를 각각 다시 선점하려고 하는 경우 발생하여 프로그램의 실행이 멈추는 현상이다.
- 때문에 Dead lock이 발생한다는 것은 프로그램을 잘못 작성한 것으로, 다양한 Dead Lock 회피 기법을 동원해서 문제를 해결해야 한다.

3.1.4. Win32 Window API & Nerv

☞ 대부분의 경우 Dead Lock 회피기법이나 Lock-Free 기법 등을 통해서 Multi Threading 문제를 해결할 수 있지만 Win3 API중에서 Windows API의 경우 Nerv와 Call Back Function 과 몇몇 가지 추가적인 문제를 일으킬 수 있다.

▶ Main Thread와 Nerv Call Back Thread의 동기화

☞ on_ virtual function에서 Main Thread Queue에 SendMessage 타입의 API를 호출하는 경우 Nerv Call Back Thread는 Main Thread가 해당 Message를 처리할 때 까지 대기함으로서 API를 호출하는 동안 Main Thread와 동기화되는 현상이 발생한다.

☞ SendMessage 타입 API

- SendMessage 는 어떤 Thread에서 호출하는가에 따라서 두 가지 방식으로 동작한다.
- SendMessage를 호출한 Thread가 Main Thread인 경우에는 다른 어떤 메시지보다 우선적으로 즉시 처리된다. 그리고 처리가 완료되면 리턴 된다.
- SendMessage를 호출한 Thread가 Main Thread가 아닌 경우에는 Main Thread Queue에 Message를 삽입한다. 그리고 Main Thread가 먼저 들어온 Message를 모두 처리한 후에 Message를 처리하고 리턴하면, SendMessage를 호출한 Thread는 SendMessage API에서 리턴 된 후에 다음 처리를 할 수 있다.
- API이름이 SendMessage가 아니더라도, SetWindowText , GetCheck,등등 HWND를 이용하여 사용하는 모든 API는 내부적으로 SendMessge를 이용하여 처리하는 것이다.

▶ Delay 문제

☞ on_ virtual function에서 SendMessage타입의 API를 호출하면 Nerv Call Back Thread는 on_ virtual function에서 Main Thread Queue에 먼저 삽입된 모든 메시지의 처리가 완료될 때까지 Delay되기 때문에 Nerv Call Back Thread에 성능 저하를 일으킬 수 있다.

▶ Dead Lock 문제

☞ Main Thread에서 Nerv의 Object나 Proxy Object의 create/destroy 함수를 호출하고, Nerv Call Back Thread의 on_ virtual function에서 SendMessage 타입의 API를 호출하는 경우 Deadlock이 발생할 확률이 발생한다.

☞ Nerv Call Back Thread는 on_ virtual functiond에서 SendMessage 타입의 API를 호출하여 처리 완료를 대기하고 있고, Main Thread는 Windows Procedure에서 먼저 들어온 다른 메시지 안에서 Nerv Object(또는 Proxy Object)를 create/destroy 하고 있는 상황에서 DeadLock 이 발생한다.

☞ Nerv Object(또는 Proxy Object)의 create/destroy method는 on_ virtual function이 실행되고 있는 동안에는 대기하기 때문이다.

▶ 문제해결 방법

☞ Nerv Call Back Thread의 on_ virtual function에서 SendMessage 타입의 API를 사용하지 않는다.

☞ SendMessage를 타입의 API를 대신하여 PostMessage API를 이용하여 프로그래밍 한다.

☞ PostMessage

- Message를 Main Thread Queue에 삽입하고 즉시 리턴 한다.
- 즉, 어떠한 대기시간도 없다.

☞ Nerv Call Back Thread의 on_ virtual function에서 Windows에 어떤 출력을 하도록 하기 위해서는 Window User Message를 만들어서 PostMessage API로 Main Thread에 Message를 전달함으로써 처리하도록 한다.

☞ 이렇게 하면 Delay 문제와 Dead Lock 문제를 모두 해결할 수 있다.

3.1.5. Thread & Nerv API

▶ Nerv 의 모든 API(Class의 Method 포함)는 어느 Thread에서 호출하여도 동작하도록 설계되어져 있다.

☞ Windows Main GUI Thread (Windows Procedure)

- 단 이 경우 몇몇 불안정 CASE가 있다. 다음 단에서 설명

☞ Main Thread

☞ Nerv Call Back Thread (on_ virtual function)

☞ Anouther All Thread

☞ Anouther All Timer

3.1.6. Windows Main GUI Thread (Windows Procedure) & Nerv API

▶ Windows Procedure는 사용자 응답시간이 실시간으로 이루어져야 하기 때문에 Nerv API를 사용하는데 있어서 주의를 필요로 한다.

☞ Window Procedure에서 호출할 때 주의 하여야 하는 API

- call: stub의 on_call의 처리에 영향을 받음
- connect: stub의 on_connect의 처리에 영향을 받음
- signal: 단 signal은 on_signal에 영향 받지 않음, ack에 따른 네트워크 부하에 영향

☞ 위 3가지 API는 limit time이라는 제한 시간 파라미터를 가지고 있어서, 시간을 소요할 가능성이 있는 API 이다.

▶ Window Procedure에서 call, signal, connect를 호출하면 대응하는 Stub에서의 처리를 모두 기다리게 된다.

☞ 버튼을 눌렀을 때 이들 API를 호출한다면 이는 해당 결과를 기다렸다가 사용자에게 응답한다는 것이다.

☞ 사용자가 기다릴 수 있고, API 처리동안 GUI가 잠시 멈추는 것을 허용할 수 있다면 무관하게 호출하여도 된다.

☞ 그렇지 않다면, Window Procedure 이벤트에서 별도의 Thread 신호를 주거나 One shout Timer(주기적이 아닌 한번만 동작하도록 지시한 타이머)를 이용하여 호출하여야 한다.

▶ Windows Procedure에서 WM_TIMER , 즉 Win32 Timer에서 이들 함수를 호출하도록 하는 것을 자제하여야 한다.

☞ Win32 Timer에서 시간을 소요하는 API를 호출하는 경우 Nerv API가 소요시간(한번에 limit time을 가지는 여러개의 API를 호출하는 경우 소요시간의 합)이 많아 지면 다른 GUI의 응답시간이 느려지게 된다.

☞ 이 경우 Win32 Timer 외의 다른 Timer를 이용하여 처리한다.

-

4. Nerv Debuging

4.1. nerv_err_t 타입 API

▶ 대부분의 Nerv API는 void , bool, nerv_err_t 타입중 하나를 리턴으로 가진다.

▶ nerv_err_t 타입 API는 Error가 발생하지 않으면 NERV_ERR_NONE_(zero,0)을 리턴하고 err가 발생하면 zero 가 아닌 다른 값을 가진다. 따라서 다음과 같은 방법으로 예외처리를 하면 편리하다.

```
if( nerv_err_t err = nerv_function() )
{
    // 예외 처리 예제
    printf("error = %s(%04X)", get_err_string(err), err);
    switch(err)
    {
    case ...:
        break;
    default:
    }
}
```

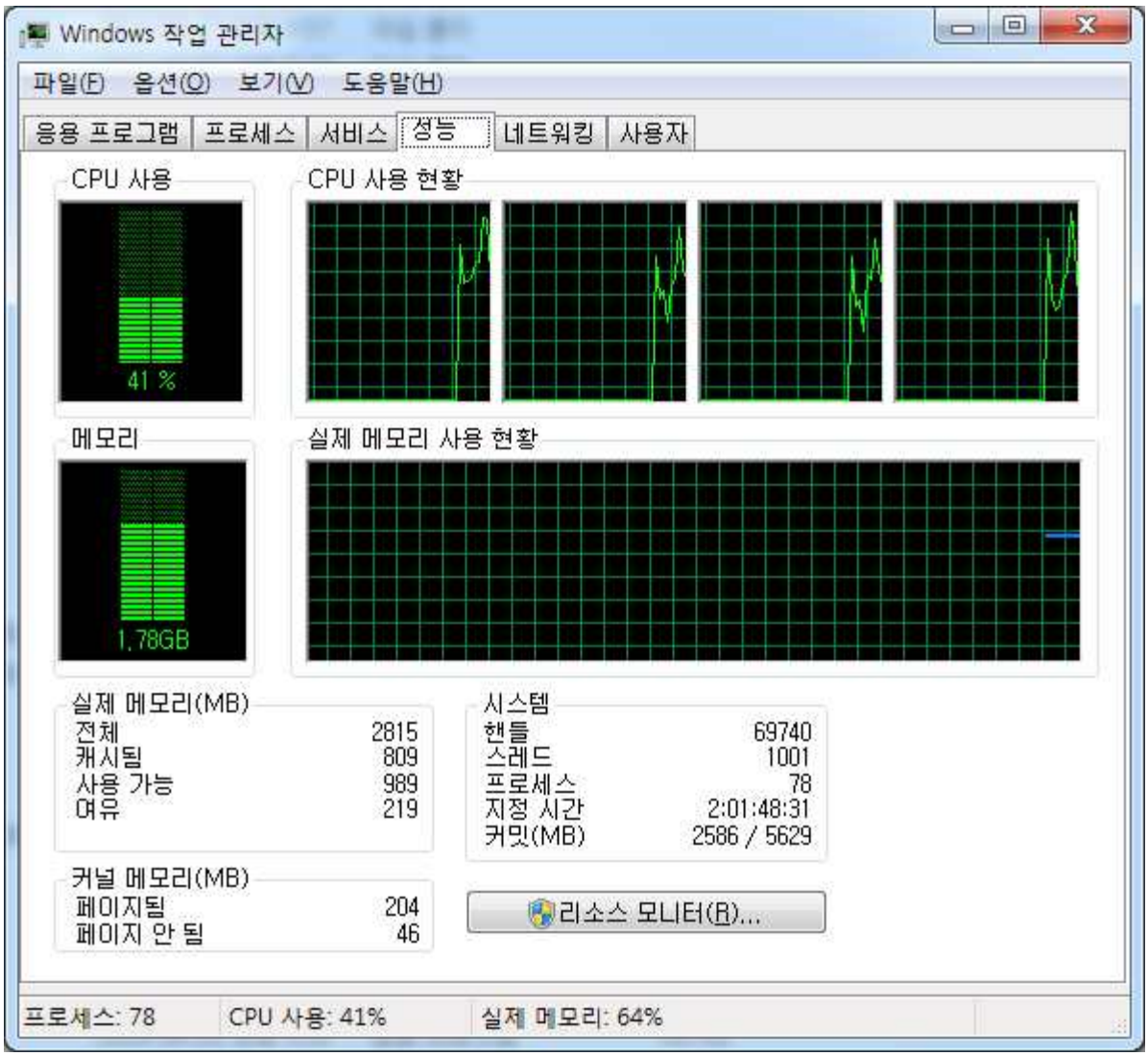
```
if( nerv_err_t err = nerv_function() )
{
    // 예외 처리 예제
}
else
{
    // 정상 처리
}
```

☞ 에러의 상수값은 nerv object framework x.x.4.1을 참조한다.

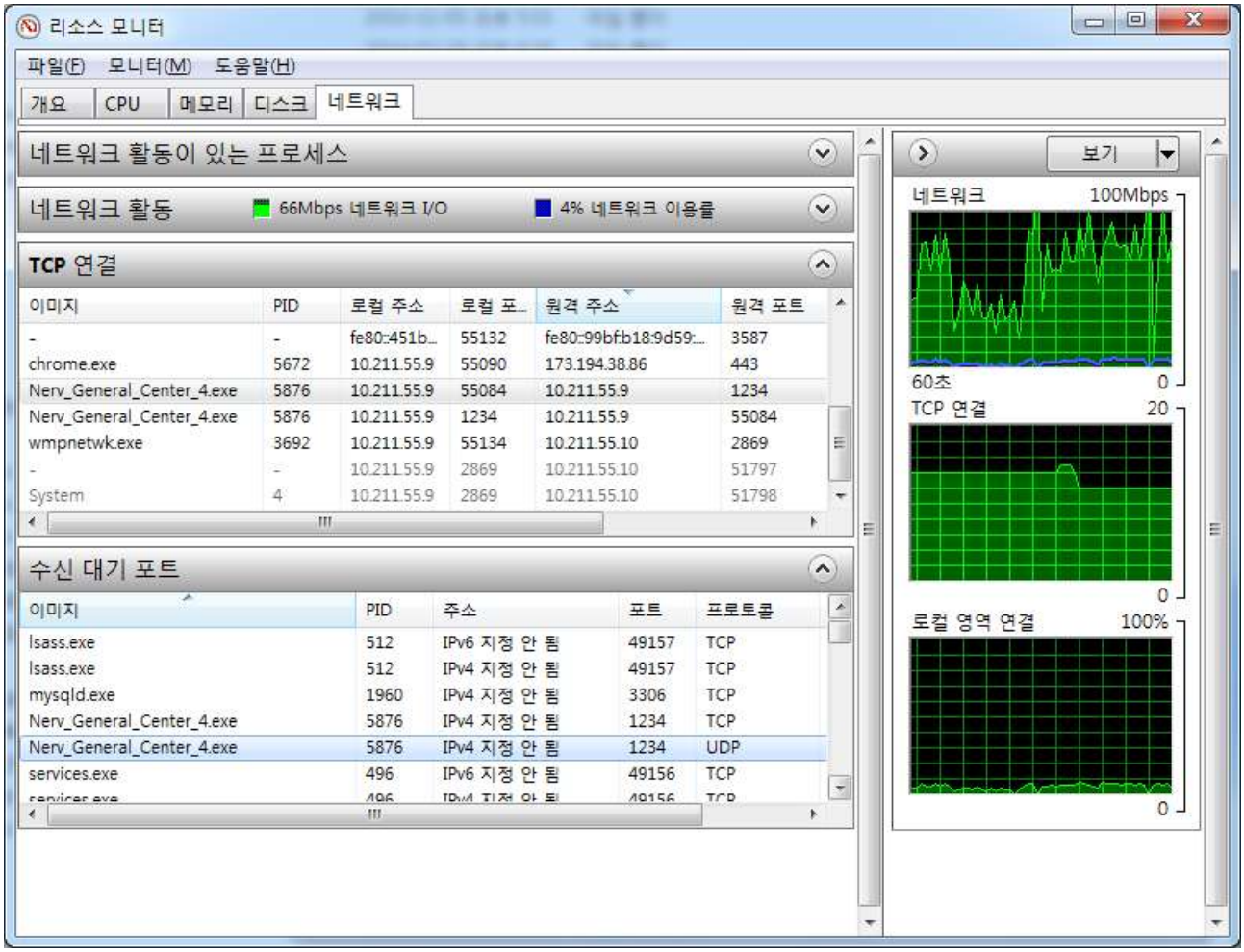
▶ 대부분의 시스템 예외 상황은 에러값을 통해서 확인할 수 있다.

4.2. 리소스 모니터

▶ 작업관리자에서 리소스 모니터를 실행하면 네트워크 상황을 모니터링 할 수 있다.



▶ 리소스 모니터



☞ [수신대기 포트]를 통해서 현재 어떤 PID의 Nerv_General_Center.exe가 어떤 Domain(Port)을 서비스하고 있는지 알수 있다.

☞ [TCP 연결]을 통해서 어떤 노드와 연결 상태인지 알 수 있다.

- [수신대기 포트]를 먼저 확인하여 Client-Server 관계를 확인할 수 있다.

☞ 통신이 완전히 되지 않는 경우 먼저 확인한다.

4.3. 방화벽

▶ 통신이 되지 않는 상황이라면 방화벽을 확인해 보아야 한다.

▶ Windows는 무의식적으로 방화벽을 켜도록 유도하므로 Nerv를 방화벽에 등록하여 주는 것이 방화벽 문제를 격지 않는 방법이다.