함정 전투체계를 위한 고가용성 알고리즘 개발

3차 보고서

연구기관

광운대학교 컴퓨터소프트웨어학과

강석중 교수

2012-11-29

목 차

[제 1 장 서론 3](#_Toc345853811)

[제 2 장 고가용성 알고리즘 테스트 4](#_Toc345853812)

[2-1 테스트 환경 4](#_Toc345853813)

[2-2 테스트 프로그램 5](#_Toc345853814)

[2-2-1 Surveillance Radar Simulation Application 5](#_Toc345853815)

[2-2-2 Track Management Application 8](#_Toc345853816)

[2-3 테스트 시나리오 12](#_Toc345853817)

[2-2-3 Failover 테스트 시나리오 12](#_Toc345853818)

[2-2-4 Recovery 테스트 시나리오 13](#_Toc345853819)

[2-2-5 성능 평가 15](#_Toc345853820)

[제 4 장 결론 16](#_Toc345853821)

[[참고문헌] 17](#_Toc345853822)

[[부록] 18](#_Toc345853823)

[고가용성 동기화 프로그램 API 리스트 18](#_Toc345853824)

[CDDS\_Sync\_Def.h 18](#_Toc345853825)

[CDDS\_Sync\_Core 클래스 18](#_Toc345853826)

[CDDS\_Sync\_Data 클래스 19](#_Toc345853827)

[CDDS\_Sync\_Error 클래스 21](#_Toc345853828)

[CDDS\_Sync\_Log 클래스 22](#_Toc345853829)

[CDDS\_Sync\_Main 클래스 23](#_Toc345853830)

[CDDS\_Sync\_MSG 클래스 30](#_Toc345853831)

[고가용 알고리즘 sequence diagram 32](#_Toc345853832)

[고가용 알고리즘 프로그래밍 가이드 37](#_Toc345853833)

[Memory Manager Design Specification 42](#_Toc345853834)

[개요 42](#_Toc345853835)

[C 인터페이스 42](#_Toc345853836)

[C++ 인터페이스 43](#_Toc345853837)

[성능 평가 43](#_Toc345853838)

[Track Generator Application Codes 44](#_Toc345853839)

[Track Management Application Codes 47](#_Toc345853840)

# 제 1 장 서론

본 연구에서는 멀티 플랫폼에서 운영 가능한 함정 전투체계의 이중화를 통한 고가용성 핵심 알고리즘의 개발을 목표로 하고 있다. 1차와 2차 보고서를 통하여 전투체계 고가용성 알고리즘과 OMG(Object Management Group) 표준의 하나인 AMSM (Application Management and System Monitoring)과의 연관성을 살펴 보았으며, OACE에 제시된 고가용성 알고리즘의 최신 경향에 대하여 살펴 보았다. 또한 본 연구에서 제시될 고가용성 알고리즘의 설계 안(class diagram, sequence diagram 등)을 도출하였다.

본 3차 보고서에서는 전투체계의 source track을 생성하는 탐색레이더 시뮬레이터와 정보처리장치에서 수행되는 track management 기능을 약식으로 구현한 테스트 프로그램을 활용하여 구현된 고가용성 알고리즘을 테스트하고, 주요 기능에 대하여 성능 평과 결과를 제시하고자 한다.

또한 3차 보고서에서는 standby와 active 어플리케이션 상태 동기화 시 기존의 콜백 함수를 이용하여 사용자 메모리 영역을 복사하는 대신에, 정해진 메모리 영역에 동기화를 위한 사용자 데이터를 배치할 수 있도록 memory management algorithm을 추가적으로 구현하여 사용자의 편의성을 증진 시켰다.

# 제 2 장 고가용성 알고리즘 테스트

## 2-1 테스트 환경

구현된 고가용성 알고리즘을 테스트하고자 그림 1과 같은 테스트 환경을 갖추었다. Source tracks을 생성하는 Surveillance Radar Simulation Application(이하 Track Generator)을 이용하여 source tracks을 생성하여, CSDB로 publish하고, 고가용성 알고리즘을 사용하는 Track Management Application에서 source tracks을 subscribe하여 system tracks을 계산한 후 Active 모드의 application은 system tracks을 publish하게 된다.



그림 . 고가용성 알고리즘 테스트 환경

이 테스트 환경에서 publish되는 system tracks 관련된 사항은 본 연구의 과제 범위가 아니므로 Publish된 system tracks은 관련 콘솔 및 TEWA 등의 모듈에서 사용되는 것으로 가정하였다.

## 2-2 테스트 프로그램

테스트 프로그램은 2가지로 나누어 진다. 첫 째는 source track을 생성하는 Surveillance Radar Simulation Application 이며, 둘 째는 수신된 source track으로부터 system track을 생성하는 Track Management Application 이다.

### Surveillance Radar Simulation Application

사용자는 화면상에서 타켓의 이동 시작점 (P0) 및 끝점 (Pn) 위치를 마우스 왼쪽 버튼을 이용하여 클릭하며, Surveillance Radar Simulation Application 는 그림 3과같이 source track을 생성하여 DDS로 전송하게 된다.

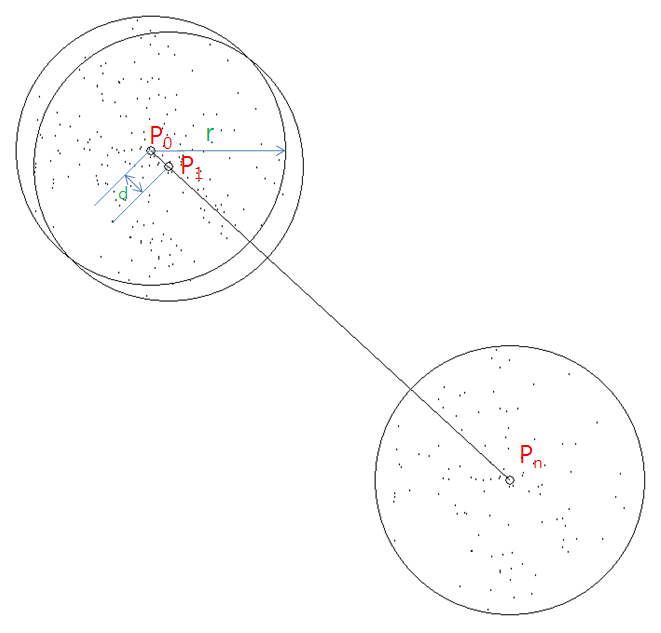


그림 . Source track generation 알고리즘

위의 Track Generator는 P0, Pn 사이에 직선의 매개 방정식을 이용하여 두 점의 거리가 d가 유지되는 n - 1개의 좌표를 생성하게 되며, 생성된 각 좌표 점에서 반지름 r 인 원내에서 100개의 점(source track)을 생성하게 된다. 100개의 source track은 각 점을 중심으로 하는 반지름 r인 원내에서 중복없이 랜덤 생성되도록 하였다. 또한 생성되는 점(source track)은 루프내에서 연속하여 생성되도록 하였으며, Pi과 Pj 사이의 이동은 surveillance radar에서 탐색한 표적 궤적을 모의하기 위하여 1초간 sleep 하도록 하였다.

프로그램 내부에서 반지름 r, 두 점 사이의 간격 d 및 P0과 Pn 사이의 생성되는 점의 개 수 n 및 그 간격을 조절할 수 있다.

이렇게 생성된 source track은 아래와 같은 source track 메시지를 DDS를 이용하여 전송하였다. 단지 Track Generator는 아래의 메시지중 키에 해당하는 필드와, 위치에 해당하는 source\_track\_number, position\_x, position\_y 3개의 필드만을 사용하였다.

|  |
| --- |
| typedef struct MSG\_TR\_SOURCE\_RADAR  {  #ifdef \_\_cplusplus  typedef struct MSG\_TR\_SOURCE\_RADARSeq Seq;  #ifndef NDDS\_STANDALONE\_TYPE  typedef MSG\_TR\_SOURCE\_RADARTypeSupport TypeSupport;  typedef MSG\_TR\_SOURCE\_RADARDataWriter DataWriter;  typedef MSG\_TR\_SOURCE\_RADARDataReader DataReader;  #endif  #endif  DDS\_HEADER header;  DDS\_Char source\_id[8];  DDS\_UnsignedShort type;  DDS\_UnsignedLong source\_track\_number;  DDS\_UnsignedShort sim\_live\_type;  DDS\_UnsignedShort lost\_status\_type;  DDS\_UnsignedShort track\_quality;  DDS\_Float position\_x;  DDS\_Float position\_y;  DDS\_Float position\_z;  DDS\_Float velocity\_x;  DDS\_Float velocity\_y;  DDS\_Float velocity\_z;  DDS\_UnsignedShort category;  DDS\_UnsignedShort identification;  DDS\_UnsignedLong time\_tag;  DDS\_Short iff\_mode\_1;  DDS\_Short iff\_mode\_2;  DDS\_Short iff\_mode\_3;  DDS\_Short iff\_mode\_4;  DDS\_UnsignedShort iff\_mode\_1\_valid;  DDS\_UnsignedShort iff\_mode\_2\_valid;  DDS\_UnsignedShort iff\_mode\_3\_valid;  DDS\_UnsignedShort iff\_mode\_4\_valid;  } MSG\_TR\_SOURCE\_RADAR; |

아래 그림 3는 최종 생성된 source track을 모두 표시한 화면이다

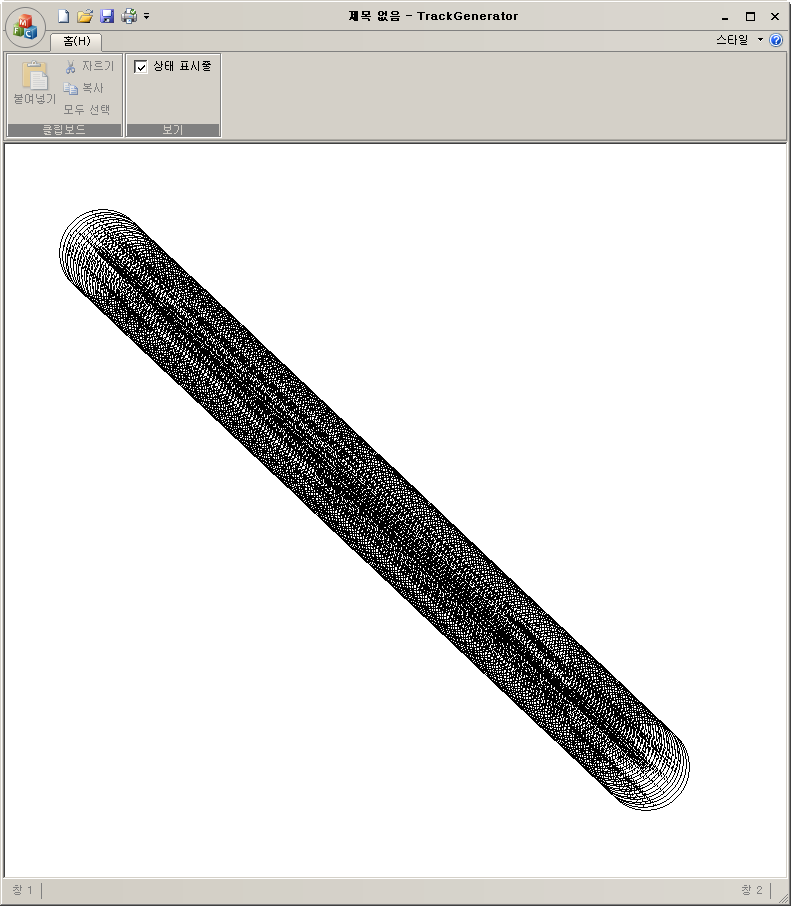


그림 . 최종 수행된 track generation 화면

### Track Management Application

모의 track management 응용프로그램은 source track을 입력으로 하여 간단한 로직으로 일정 거리내의 source track을 결합하여 하나의 system track을 생성한 후 이를 DDS로 전송하는 프로그램이다. 이 응용프로그램의 간략한 동작은 그림 4 의 activity diagram을 참조하기 바란다.



그림 . Activity diagram of track management application

Track management application은 본 연구과제 산출물인 고가용성 알고리즘을 통하여 active 또는 standby 상태를 갖는다. Active 상태의 프로그램과 standby 상태의 응용프로그램은 장애 복구 시 서로의 상태를 동기화 시키는 절차를 수행해야 한다.

동기화를 수행하는 방법은 크게 2가지로 구분된다. 동기화 콜백 함수를 사용하는 방법과 메모리 매니저 내에서 수행하는 2 가지 방법으로 구분될 수 있다.

#### 프로그램의 실행

Active 또는 standby 어떤 프로그램을 먼저 실행 시켜도 상관은 없으나 active와 standby간 주고 받을 메시지는 파라메터로 전달해야 한다. 따라서 active와 standby 사이의 파라메터 값은 서로 반대가 되어야 한다.

한 노드에서 아래와 같이 실행을 시키는 경우,

|  |
| --- |
| C:\>RecvDDS\_Track.exe **PS1 PS2** |

그 반대쪽은 아래와 같이 실행을 시켜야 한다.

|  |
| --- |
| C:\>RecvDDS\_Track.exe **PS2 PS1** |

이와 같이 파라메터 값은 standby 모드로 수행되는 프로그램이 active 모드로 변경된다고 하더라도 동일한 파라메터 값을 사용해야 한다. 즉, 내부적으로 주어진 파라메터를 이용하여 heartbeat을 주고 받기 위한 topic을 생성하므로 이 파라메터 값이 변경되거나 하면 안 된다. 이와 관련된 메소드는 CDDS\_Sync\_Main::StartSync() 함수를 참조한다.

#### 동기화 콜백 사용 방법

또한 동기화 콜백 함수는 상태 공유 변수 전송을 위한 전송 콜백 함수와 수신된 상태 공유 변수를 이용하여 standby 응용프로그램의 상태를 active와 동기 시키는 수신 콜백 함수로 구분된다.

|  |
| --- |
| static void Callback\_Send(void\*)  {  DWORD dwSize = 1024\*1024; // is more than MAX\_TRACK \* sizeof(SystemTrack\_t) + sizeof(int) of \*gpnSysTrack  char \*pSendData = new char[dwSize]; // malloc enough big buffer for gpnSysTrack & gpSysTrack  sync\_lock();  memcpy(pSendData, gpnSysTrack, sizeof(int));  memcpy(pSendData + sizeof(int), gpSysTrack, sizeof(SystemTrack\_t) \* \*gpnSysTrack);  sync\_unlock();  if( !CDDS\_Sync\_Main::getInstance().SetSendData(pSendData, sizeof(int) + sizeof(SystemTrack\_t) \* \*gpnSysTrack) )  {  printf("Error SetSendData\n");  }  delete[] pSendData;  } |

그림 . 전송 동기화 콜백의 예

|  |
| --- |
| static void Callback\_Receive(void\*)  {  DWORD dwSize = CDDS\_Sync\_Main::getInstance().GetReceiveDataSize();    if( dwSize <= 0 )  return;  char \*pSendData = new char[dwSize];  if( CDDS\_Sync\_Main::getInstance().GetReceiveData(pSendData) )  {  printf("Error GetReceiveData\n");  }  sync\_lock();  memcpy(gpnSysTrack, pSendData, sizeof(int));  memcpy(gpSysTrack, pSendData + sizeof(int), sizeof(SystemTrack\_t) \* \*gpnSysTrack);  sync\_unlock();  delete[] pSendData;  } |

그림 . 수신 동기화 콜백의 예

동기화 콜백 함수들은 상태 동기화 동안 sync\_lock/sync\_unlock 메커니즘을 이용하여 상태 공유 변수의 무결성을 지켜야 한다. 보통의 경우 이러한 critical section 내에서의 행위는 최소한의 시간에 이루어지도록 구현되어야 한다. 상기와 같은 memory copy는 복사할 영역의 크기에 따라 사용되는 system clock cycle이 차이가 나지만 대략 수십개 또는 수백개 내외 이므로 보통의 경우 0.01초 내에 충분히 수행되므로 전반적으로 system에 미치는 overhead는 무시할 수 있는 수준으로 볼 수 있다.

#### 메모리 매니저 사용 방법

상태 공유 변수를 본 연구과제 산출물인 메모리 매니저의 버퍼 영역안에 존재하는 경우 동기화가 필요한 시점에 메모리 매니저 내에서 자동으로 동기화를 수행하게 된다.

|  |
| --- |
| // initialize memory manager  mm\_init();  **.**  **.**  **.**  // memory allocation for synchronized area  gpSysTrack = (SystemTrack\_t\*)mm\_malloc(sizeof(SystemTrack\_t) \* MAX\_TRACK); // 16 \* 500 = 80,000 bytes  gpnSysTrack = (int \*)mm\_alloc(sizeof(int)); |

그림 . 메모리 매니저를 이용한 동기화 영역 할당

## 2-3 테스트 시나리오

테스트 시나리오는 고가용성 알고리즘의 주요 2 가지 기능인 장애절체(failover)와 복구(recovery)의 정상적인 동작을 테스트하기 위하여 다음과 같은 2가지 시나리오를 가정하였다. 2가지 시나리오가 동작하기 위해서 본 과제의 주요 성능 평가 항목인 failover time와 recovery time에 대하여 정의할 필요가 있다. 고가용성 시스템에서 일반적으로 다음과 같이 정의된다.

장애절체(failover) time: active 노드에서 발생한 장애를 standby 노드에서 검출하여 standby 노드가 active 노드의 역할을 하기까지 소요되는 시간

복구(recovery) time: 장애를 일으킨 active 노드가 적절한 조치를 취한 후 정상적인 동작을 취하기 위하여 준비하기까지 소요되는 시간

### Failover 테스트 시나리오

Active application을 갑작스럽게 kill 시켜 standby application이 정상적으로 active 상태로 동작하는지를 확인하기 위한 테스트이다.

그림 . 장애절체 테스트 시나리오

에서 Active는 3, 4, 6 단계 어느 곳에서도 장애를 만날 수 있다. 장애를 만나는 경우 Standby는 heartbeat 등의 내부 알고리즘에 의해서 장애를 검출(detect failure)하게 된다. 장애를 검출한 후 active의 역할을 대신하기까지 소요되는 시간을 로그를 통하여 계산을 한다. 장애 검출은 heartbeat를 이용하기 때문에 이 heartbeat 주기 및 간격은 장애검출에 매우 큰 영향을 미치게 된다.

### Recovery 테스트 시나리오

중단된 active application을 재기동 시켜 standby 상태로 운영되는지를 확인한다. Standby 상태로 정상적으로 운영이 되기 위해서는 현재 active 상태로 동작중인 application과 내부 공유 영역의 상태가 동기화 되어야 한다.

그림 . 복구 시나리오

에서 active는 장애를 복구한 후 (correct failure) 현재 active의 역할을 대신하고 있는 과거의 standby에게 동기화 데이터를 요청하여 active와 현재의 상태를 맞추게 된다. 이 때 소요되는 시간은 내부 동기화 데이터의 크기에 비례하여 커지게 된다. 또한 CSDB 네트워크의 상태에 따라서도 영향을 받게 된다.

### 성능 평가

위의 failover와 recovery time을 측정은 “장애절체 및 복구 sequence diagram”를 참조한다. Failover와 recovery 소요 시간 응용프로그램의 특성에 매우 종속적이나, 기본적인 특징을 살펴 보면 다음과 같다.

Failover 시간은 일반적으로 평균 heartbeat 전송 주기 x 처리되지 않은 CS DDS 메시지 개 수[[1]](#footnote-1)로 정의되며, Recovery 시간은 상태 동기 데이터의 크기와 네트워크 전송 속도에 따라 결정된다. 즉, 상태 동기 데이터의 크기가 작고, 네트워크 전송 속도가 빠를수록 recovery 시간은 짧아지게 된다.

# 제 4 장 결론

본 연구에서는 고가용성 알고리즘을 이루는 핵심 알고리즘과 더불어 멀티 플랫폼에서 운영될 수 있도록 OSAL 에 대한 연구를 수행하고 알고리즘 설계의 한 방법으로 클래스 다이어그램 등의 설계안을 제안하였다. 고가용성 알고리즘 구현에 있어서 중점을 두고 있는 사항은 다음과 같다.

* Active와 Standby 프로세스는 전투체계를 이루는 DDS를 통하여 통신을 수행하여 별도의 TCP 또는 UDP 통신을 수행하지 않는다. 따라서 고가용성 알고리즘 운영시 통신 프로토콜의 혼재를 방재할 수 있다.
* Active와 Standby간 고가용성 기능을 최대한 손 쉽게 사용할 수 있도록 하였다. 또한 고가용성 알고리즘 구동 시 빚어지는 성능의 저하를 최대한 방지하도록 노력하였다. 이를 위하여 고가용서 알고리즘을 사용하는 프로세스는 동기화를 필요로 하는 부분을 정의하여 사용하게 함으로서 동기화 시 소요되는 시간을 최대한 단출할 수 있도록 하였다.
* 고가용성 알고리즘이 멀티 플랫폼 환경에서 운영될 수 있도록 가능한 OACE에서 권고하는 POSIX API를 가능한 이용하였으며, MS Windows와 같이 제한된 POSIX를 제공하는 환경에서는 일부 누락된 API를 구현하여 제공하였다.

3차 보고서에서는 설계된 고가용성 알고리즘을 구현하고, 테스트를 수행하여 그 실험 결과를 보이도록 할 계획이다. 이를 통하여 몇 가지 테스트 프로그램을 개발하고 이 테스트 결과가 향후 고가용성 알고리즘을 사용하는 참조점으로 사용될 수 있도록 계획하고 있다.

# [참고문헌]

[1] Open Architecture Computing Environment Technologies and Standards, Version 1.0, 2004

[2] Open Architecture Computing Environment Design Guidance Version 1.0, 2004

[3] Fault Tolerant CORBAR Version 1.0, 2010

[4] The Open Group Base Specification Issue 7 IEEE Std 1003.1, 2008

[5] Application Management and System Monitoring for CMS Systems Version 1.1

[6] UML Distilled, Martin Fowler, Edition 3

# [부록]

2차 연구보고서의 작성 이후 사용자의 편리성을 높이기 위하여 추가된 memory manager에 대한 설계 부분과 개발자의 향후 프로그래밍 시 도움이 될 수 있도록 본 테스트에 사용된 2개의 application에 대하여 주요 부분에 대한 source code에 대하여 기술한다

## 고가용성 동기화 프로그램 API 리스트

본 연구를 통하여 개발된 전투체계 임베디드 시스템을 위한 고가용성 알고리즘을 사용하기 위한 인터페이스는 다음과 같다.

### CDDS\_Sync\_Def.h

CDDS\_Sync\_Def.h 파일은 고가용성 알고리즘 내부에서 사용되는 값에 대한 정의를 담고 있다. 알고리즘 운영에 밀접한 active 와 standby 상태 전환 시간 및 heartbeat 전송 주기는 아래와 같이 정의되어 있다.

|  |  |  |
| --- | --- | --- |
| **정의** | **값** | **비고** |
| DDS\_SYNC\_CHANGE\_MODE\_TIME | 10000 | active 와 standby 상태 전환 시간으로 10초 |
| DDS\_SYNC\_ACTIVE\_HEARTBEAT\_TIME | 1000 | heartbeat 전송 주기로 1초 |

### CDDS\_Sync\_Core 클래스

동기화 관련 클래스에서 사용 가능한 utility 함수를 제공하는 클래스. 현재 사용자 데이터를 동기화를 위해서 복사하는 경우 필요한 locking facility를 제공한다.

#### CDDS\_Sync\_Core

기본 생성자

|  |
| --- |
| **인터페이스** |
| CDDS\_Sync\_Core |

#### sync\_lock

mutual exclusion을 위한 lock 함수

|  |  |
| --- | --- |
| **인터페이스** | |
| int sync\_lock() | |
| **Parameters 목록** | |
| void |  |
| **Return 값** | |
| int | 1: always |

#### sync\_unlock

lock 해제 함수

|  |  |
| --- | --- |
| **인터페이스** | |
| int sync\_unlock() | |
| **Parameters 목록** | |
| void |  |
| **Return 값** | |
| int | 1: always |

### CDDS\_Sync\_Data 클래스

동기화를 위해 데이터를 송수신하도록 설계된 클래스로 내부에 DDSDataReaderListener를 상속받아 사용한다.

#### CDDS\_Sync\_Data

CDDS\_Sync\_Data 생성자

|  |
| --- |
| **인터페이스** |
| CDDS\_Sync\_Data |

#### on\_data\_available

DDS 메시지가 읽기 가능해진 경우 호출되는 콜백 함수이다.

|  |  |
| --- | --- |
| **인터페이스** | |
| void on\_data\_available(DDSDataReader \*reader) | |
| **Parameters 목록** | |
| DDSDataReader \*reader | DDS를 통해 전송된 데이터 정보 |
| **Return 값** | |
| void |  |

#### CALLBACK\_DataProcess

Read 된 데이터가 Active가 보낸 Memory 동기화 정보인지 확인하여 처리하는 함수

|  |  |
| --- | --- |
| **인터페이스** | |
| void CALLBACK\_DataProcess(void\* lpInfo) | |
| **Parameters 목록** | |
| void\* lpInfo | 전송된 DDS 정보를 DDS\_SEQUENCE로 Pasing 한 Data 중 하나 |
| **Return 값** | |
| void |  |

#### ReceiveData

DDS를 통해 전송된 데이터를 처리하는 함수

|  |  |
| --- | --- |
| **인터페이스** | |
| void ReceiveData(DDSDataReader \*reader) | |
| **Parameters 목록** | |
| DDSDataReader \*reader | on\_data\_available 를 통해 bypass 된 DDS를 통해 전송된 데이터 정보 |
| **Return 값** | |
| void |  |

#### WriteData

DDS를 통해 데이터를 전송하는 함수

|  |  |
| --- | --- |
| **인터페이스** | |
| DDS\_ReturnCode\_t WriteData(DDS\_Sync\_Data\_Type \*pMsgDump) | |
| **Parameters 목록** | |
| DDS\_Sync\_Data\_Type \*pMsgDump | 전송할 정보 |
| **Return 값** | |
| DDS\_ReturnCode\_t | DDS manual 참조 |

#### SetDataWriter

Writer를 초기화하는 함수

|  |  |
| --- | --- |
| **인터페이스** | |
| bool SetDataWriter(char \*pTopicName, DDSDomainParticipant \*participant) | |
| **Parameters 목록** | |
| char \*pTopicName | Data를 전송할 Topic Name |
| DDSDomainParticipant \*participant | 참여할 도메인 클래스 |
| **Return 값** | |
| bool | true: 성공  false: 실패 |

#### SetDataReader

Reader를 초기화하는 함수

|  |  |
| --- | --- |
| **인터페이스** | |
| bool SetDataReader(char \*pTopicName, DDSDomainParticipant \*participant) | |
| **Parameters 목록** | |
| char \*pTopicName | Data가 수신될 Topic Name |
| DDSDomainParticipant \*participant | 참여할 도메인 클래스 |
| **Return 값** | |
| bool | true: 성공  false: 실패 |

#### SetMain

CDDS\_Sync\_Main 을 설정하는 함수

|  |  |
| --- | --- |
| **인터페이스** | |
| void SetMain(CDDS\_Sync\_Main \*pParent) | |
| **Parameters 목록** | |
| CDDS\_Sync\_Main \*pParent | CDDS\_Sync\_Data를 Member로 가지는 부모클래스 |
| **Return 값** | |
| void |  |

### CDDS\_Sync\_Error 클래스

내부에서 발생하는 에러를 저장하는 클래스

#### ~CDDS\_Sync\_Error

CDDS\_Sync\_Error 파괴자

|  |
| --- |
| **인터페이스** |
| ~CDDS\_Sync\_Error(void) |

#### CDDS\_Sync\_Error

CDDS\_Sync\_Error 생성자

|  |
| --- |
| **인터페이스** |
| CDDS\_Sync\_Error(void) |

#### getInstance

CDDS\_Sync\_Error Instance를 얻는다.

|  |  |
| --- | --- |
| **인터페이스** | |
| static CDDS\_Sync\_Error& getInstance() | |
| **Parameters 목록** | |
| void |  |
| **Return 값** | |
| CDDS\_Sync\_Error& | 사용될 CDDS\_Sync\_Error Class |

#### SetLastError

발생된 에러를 출력하기 위해 설정하는 함수

|  |  |
| --- | --- |
| **인터페이스** | |
| bool SetLastError(char \*cErrorMsg) | |
| **Parameters 목록** | |
| char \*cErrorMsg | 에러로 출력될 에러문장 |
| **Return 값** | |
| Bool | 에러문장이 없다면 false, 정상완료면 true |

#### GetLastError

발생된 에러를 반환하기 위한 함수

|  |  |
| --- | --- |
| **인터페이스** | |
| bool GetLastError(char \*cErrorMsg) | |
| **Parameters 목록** | |
| char \*cErrorMsg | 발생된 에러문장 |
| **Return 값** | |
| bool | \*cErrorMsg 가 Null 이면 false, 정상완료면 true |

### CDDS\_Sync\_Log 클래스

Process 진행을 Log로 출력하고 File로 저장하는 클래스

#### CDDS\_Sync\_Log

CDDS\_Sync\_Log 생성자

|  |
| --- |
| **인터페이스** |
| CDDS\_Sync\_Log() |

#### ~CDDS\_Sync\_Log

CDDS\_Sync\_Log 파괴자

|  |
| --- |
| **인터페이스** |
| ~CDDS\_Sync\_Log() |

#### static CDDS\_Sync\_Log& getInstance()

CDDS\_Sync\_Log Instance를 얻는다.

|  |  |
| --- | --- |
| **인터페이스** | |
| static CDDS\_Sync\_Log& getInstance() | |
| **Parameters 목록** | |
| void |  |
| **Return 값** | |
| CDDS\_Sync\_Log& | 사용될 CDDS\_Sync\_Log Class |

#### OpenLog

Log가 저장될 파일을 설정하는 함수

|  |  |
| --- | --- |
| **인터페이스** | |
| bool OpenLog(char \*pFileName) | |
| **Parameters 목록** | |
| char \*pFileName | 저장될 경로와 파일명 |
| **Return 값** | |
| bool | 파일 생성 실패면 false, 성공이면 true |

#### PrintLog

Log를 출력하는 함수

|  |  |
| --- | --- |
| **인터페이스** | |
| bool PrintLog(char \*afmt,...) | |
| **Parameters 목록** | |
| char \*afmt | Printf 파라메터와 동일,  단, c(문자), s(문자열), d(정수형) 만 출력가능 |
| … |  |
| **Return 값** | |
| bool | 출력할 파일이 없으면 false, 성공이면 true |

### CDDS\_Sync\_Main 클래스

동기화를 위한 Main이 되는 클래스로 이 클래스를 통해서 DDS로부터 모든 전송된 Data를 처리하도록 한다

#### CDDS\_Sync\_Main

CDDS\_Sync\_Main 생성자

|  |
| --- |
| **인터페이스** |
| CDDS\_Sync\_Main |

#### ~CDDS\_Sync\_Main

CDDS\_Sync\_Main 파괴자

|  |
| --- |
| **인터페이스** |
| ~CDDS\_Sync\_Main |

#### getInstance

CDDS\_Sync\_Main 클래스를 얻는다.

|  |  |
| --- | --- |
| **인터페이스** | |
| static CDDS\_Sync\_Main& getInstance() | |
| **Parameters 목록** | |
| void |  |
| … | |
| **Return 값** | |
| CDDS\_Sync\_Main& | 사용될 CDDS\_Sync\_Main |

#### StartSync

동기화를 시작 하는 함수. 이 함수를 통해서 동기화용 Read, Write Topic을 생성한다. 이 토픽은 active와 standby간 heartbeat로 사용된다. 이와 같이 topic name을 사용자로부터 입력 받는 이유는 전투체계 환경이 DDS 데이터버스를 이용하기 때문에 동일한 heartbeat topic을 사용함으로서 발생하는 망내의 이중화 응용프로그램 사이의 혼란을 방지하기 위함이다.

|  |  |
| --- | --- |
| **인터페이스** | |
| bool StartSync(char \*pReaderTopicName, char \*pWriteTopicName) | |
| **Parameters 목록** | |
| char \*pReaderTopicName | Reader에 사용될 Topic Name |
| char \*pWriteTopicName | Writer에 사용될 Topic Name |
| **Return 값** | |
| bool | Reader 또는 Writer topic 생성에 실패하면 false, 성공하면 true |

#### GetStatus

CDDS\_Sync\_Main의 Sync Process 생태를 반환하는 함수

|  |  |
| --- | --- |
| **인터페이스** | |
| PLAY\_STATUS GetStatus() | |
| **Parameters 목록** | |
| void |  |
| **Return 값** | |
| PLAY\_STATUS | Sync Process 상태를 나타낸다.  DDS\_SYNC\_STATUS\_IDLE : 동기화 하지 않고 있는 상태  DDS\_SYNC\_STATUS\_SYNCHRONIZING : 동기화 중인 상태  DDS\_SYNC\_STATUS\_QUEUE : 동기화가 끝나고 동기화 중에 들어 온 Message Queue를 처리 중인 상태 |

#### SetStatus

CDDS\_Sync\_Main의 Sync Process 생태를 설정하는 함수

|  |  |
| --- | --- |
| **인터페이스** | |
| void SetStatus(PLAY\_STATUS iStatus) | |
| **Parameters 목록** | |
| PLAY\_STATUS iStatus | Sync Process 상태를 나타낸다.  DDS\_SYNC\_STATUS\_IDLE : 동기화 하지 않고 있는 상태  DDS\_SYNC\_STATUS\_SYNCHRONIZING : 동기화 중인 상태  DDS\_SYNC\_STATUS\_QUEUE : 동기화가 끝나고 동기화 중에 들어 온 Message Queue를 처리 중인 상태 |
| **Return 값** | |
| void |  |

#### WaitProcessing

CDDS\_Sync\_Main 이 Active 또는 Standby가 결정될 때까지 대기하거나,

Status가 DDS\_SYNC\_STATUS\_QUEUE 인 경우 Message Queue를 처리하는 함수

|  |  |
| --- | --- |
| **인터페이스** | |
| void WaitProcessing() | |
| **Parameters 목록** | |
| void |  |
| **Return 값** | |
| void |  |

#### StandbyMsgProcessing

Standby Mode 에서 Active Mode 로 바뀔 때 Standby 와 Active 바뀌는 시간 동안 들어온 메시지를 처리하는 함수

|  |  |
| --- | --- |
| **인터페이스** | |
| void StandbyMsgProcessing() | |
| **Parameters 목록** | |
| void |  |
| **Return 값** | |
| void |  |

#### DeleteStandbyMsg

StandbyMsgProcessing()에서 처리하기 위해서 일정시간동안 들어 오는 메시지를 Queue에 담는데 이때 일정시간 이상을 넘긴 메시지를 삭제하는 함수

|  |  |
| --- | --- |
| **인터페이스** | |
| void DeleteStandbyMsg(DDS\_Time\_t srcTime) | |
| **Parameters 목록** | |
| DDS\_Time\_t srcTime | DDS Message가 전송한 Data 전송시간 |
| **Return 값** | |
| void |  |

#### MessageRecive

동기화를 하기 위해 사용자가 설정한 Receive Callback 함수를 실행하는 함수

|  |  |
| --- | --- |
| **인터페이스** | |
| Void MessageRecive(int iType, void\* pMsg, DWORD dwSize, DDS\_SequenceNumber\_t seqNum, DDS\_Time\_t srcTime) | |
| **Parameters 목록** | |
| int iType | 동기화를 위한 Callback 함수의 Type으로 SetCallback 시 반환되는 값으로 사용한다. |
| void\* pMsg | DDS로 부터 전송된 Sequence Data |
| DWORD dwSize | pMsg의 Size |
| DDS\_SequenceNumber\_t seqNum | DDS로 부터 전송된 Sequence Number |
| DDS\_Time\_t srcTime | DDS로 부터 전송된 전송 시간 |
| **Return 값** | |
| void |  |

#### SetCallback

함수를 이용하여 상태 데이터를 동기화 하는 경우 동기화 함수를 등록하는 함수이다. 동기화 함수는 사용자 영역 데이터를 고가용성 라이브러리로 복사하는 역할을 수행하게 된다.

|  |  |
| --- | --- |
| **인터페이스** | |
| Int SetCallback(CALLBACK\_Receive\_FUNC\_Sync func) | |
| **Parameters 목록** | |
| CALLBACK\_Receive\_FUNC\_Sync func | 동기화 하기 위해 사용되는 Callback 함수를 설정 |
| **Return 값** | |
| int | 설정된 Type Index를 반환한다. |

#### SetDataCallback

동기화 할 데이터를 Send, Revevie할 Callback 함수를 설정한다

|  |  |
| --- | --- |
| **인터페이스** | |
| boolSetDataCallback(CALLBACK\_FUNC\_Sync funcSendData, CALLBACK\_FUNC\_Sync funcReceiveData) | |
| **Parameters 목록** | |
| CALLBACK\_FUNC\_Sync funcSendData | 동기화 데이터를 Send시에 사용될 Callback 함수 |
| CALLBACK\_FUNC\_Sync funcReceiveData | 동기화 데이터를 Receive시에 사용될 Callback 함수 |
| **Return 값** | |
| bool | 항상 true 반환 |

#### SendSyncData

동기화 할 데이터를 Standby에 전송하는 함수

|  |  |
| --- | --- |
| **인터페이스** | |
| boolSendSyncData() | |
| **Parameters 목록** | |
| void |  |
| **Return 값** | |
| bool | 실패하면 false, 성공하면 true |

#### ReceiveSyncData

동기화 할 Data를 Active로부터 받는 함수

|  |  |
| --- | --- |
| **인터페이스** | |
| boolReceiveSyncData(DDS\_Sync\_Data\_Type\* pDDSData) | |
| **Parameters 목록** | |
| DDS\_Sync\_Data\_Type\* pDDSData | 동기화 Data |
| **Return 값** | |
| bool | 실패하면 false, 성공하면 true |

#### GetMode

현재 Active 인지 Standby 인지 아직 설정 되지 않았는지 반환하는 함수. 사용자는 이 함수에서 리턴된 결과에 따라 active 또는 standby 에서 수행할 로직을 결정해야 한다. 예를 들면, active 모드의 어플리케이션은 계산된 결과는 DDS를 통해서 전송을 하지만 standby 모드의 어플리케이션은 계산된 결과를 전송하지 않아야 한다.

|  |  |
| --- | --- |
| **인터페이스** | |
| PLAY\_MODE GetMode() | |
| **Parameters 목록** | |
| void |  |
| **Return 값** | |
| PLAY\_MODE | 현재 Mode를 반환한다.  DDS\_SYNC\_MODE\_NONE: 설정되지 않은 상태. DDS를 통해 Message가 전송되도 처리하지 않고 대기한다.  DDS\_SYNC\_MODE\_ACTIVE: Active로 설정된 상태. DDS를 통해 Message가 전송되면 Active 처리 Process를 진행한다.  DDS\_SYNC\_MODE\_STANDBY: Standby로 설정된 상태. DDS를 통해 Message가 전송되면 Standby 처리 Process를 진행한다. |

#### SetMode

현재 Mode를 설정한다.

|  |  |
| --- | --- |
| **인터페이스** | |
| void SetMode(PLAY\_MODE PLAY\_MODE) | |
| **Parameters 목록** | |
| PLAY\_MODE PLAY\_MODE | 현재 Mode를 설정한다.  DDS\_SYNC\_MODE\_NONE: 설정되지 않은 상태. DDS를 통해 Message가 전송되도 처리하지 않고 대기한다.  DDS\_SYNC\_MODE\_ACTIVE: Active로 설정된 상태. DDS를 통해 Message가 전송되면 Active 처리 Process를 진행한다.  DDS\_SYNC\_MODE\_STANDBY: Standby로 설정된 상태. DDS를 통해 Message가 전송되면 Standby 처리 Process를 진행한다. |
| **Return 값** | |
| void |  |

#### GetLastError

에러가 있었다면 마지막 에러를 반환한다

|  |  |
| --- | --- |
| **인터페이스** | |
| boolGetLastError(char \*pErrorMsg) | |
| **Parameters 목록** | |
| char \*pErrorMsg | 에러 문장 |
| **Return 값** | |
| bool | \*cErrorMsg 가 Null 이면 false, 정상완료면 true |

#### OpenLog

파일로 저장될 Log를 설정한다.

|  |  |
| --- | --- |
| **인터페이스** | |
| bool OpenLog(char \*pFileName) | |
| **Parameters 목록** | |
| char \*pFileName | 저장될 Log File Name |
| **Return 값** | |
| bool | LogFile 을 생성에 실패하면 false, 성공하면 true |

#### SetSendData

동기화 전송을 위해 송신 데이터를 복사하는 함수

|  |  |
| --- | --- |
| **인터페이스** | |
| bool SetSendData(char \*pSendData, DWORD dwSize) | |
| **Parameters 목록** | |
| char \*pSendData | 전송할 Data |
| DWORD dwSize | 전송할 Data Size |
| **Return 값** | |
| bool | 복사에 실패하면 false, 성공하면 true |

#### GetReceiveDataSize

전송된 동기화 Data Size 반환

|  |  |
| --- | --- |
| **인터페이스** | |
| DWORD GetReceiveDataSize() | |
| **Parameters 목록** | |
| void |  |
| **Return 값** | |
| DWORD | 전송된 동기화 Data Size를 반환 |

#### GetReceiveData

전송된 Data 반환

|  |  |
| --- | --- |
| **인터페이스** | |
| bool GetReceiveData(char \*pReceive) | |
| **Parameters 목록** | |
| char \*pReceive | 전송된 Data 반환 |
| **Return 값** | |
| bool | pReceive가 Null이거나 전송된 Data의 Size가 0 보다 작으면 false, 성공하면 true |

#### StartChangeModeTimer

Standby mode인 경우 일정 시간 동안 Active 로부터 신호가 없으면 Active Mode로 전환되도록 하는 Timer를 시작하는 함수

|  |  |
| --- | --- |
| **인터페이스** | |
| Int StartChangeModeTimer() | |
| **Parameters 목록** | |
| void |  |
| **Return 값** | |
| int | Timer 설정이 실패하면 false, 성공하면 true |

#### SetLastSeqNum

마지막에 전송된 Sequence Number 설정

|  |  |
| --- | --- |
| **인터페이스** | |
| void SetLastSeqNum(DDS\_SequenceNumber\_t LastSeqNum) | |
| **Parameters 목록** | |
| DDS\_SequenceNumber\_t LastSeqNum | 마지막에 들어온 Sequence Number |
| **Return 값** | |
| void |  |

### CDDS\_Sync\_MSG 클래스

동기화를 위한 Message를 처리하는 클래스로 이 클래스를 통해서 Active가 보내는 Heartbeat, Active가 처리한 Sequence Number, 동기화 요청, 동기화 완료 등을 처리한다.

#### CDDS\_Sync\_MSG

CDDS\_Sync\_MSG 생성자

|  |
| --- |
| **인터페이스** |
| CDDS\_Sync\_MSG |

#### ~CDDS\_Sync\_MSG

CDDS\_Sync\_MSG 파괴자

|  |
| --- |
| **인터페이스** |
| ~CDDS\_Sync\_Main |

#### on\_data\_available

DDS Reader를 통해 들어온 메시지를 받아주는 함수

|  |  |
| --- | --- |
| **인터페이스** | |
| void on\_data\_available(DDSDataReader \*reader) | |
| **Parameters 목록** | |
| DDSDataReader \*reader | DDS를 통해 전달된 데이터 |
| **Return 값** | |
| void |  |

#### CALLBACK\_MSGProcess

Read 된 데이터가 어떤 Message 데이터인지 확인하여 처리하는 함수

|  |  |
| --- | --- |
| **인터페이스** | |
| void CALLBACK\_MSGProcess(void\* lpInfo) | |
| **Parameters 목록** | |
| void\* lpInfo | 전송된 DDS 정보를 DDS\_SEQUENCE로 Pasing 한 Data 중 하나 |
| **Return 값** | |
| void |  |

#### ReceiveMSG

DDS 를 통해 들어온 데이터를 처리하는 함수

|  |  |
| --- | --- |
| **인터페이스** | |
| void ReceiveMSG(DDSDataReader \*reader) | |
| **Parameters 목록** | |
| DDSDataReader \*reader | on\_data\_available 를 통해 bypass 된 DDS를 통해 전송된 데이터 정보 |
| **Return 값** | |
| void |  |

#### WriteMSG

DDS를 통해 데이터를 전송하는 함수

|  |  |
| --- | --- |
| **인터페이스** | |
| DDS\_ReturnCode\_t WriteMSG(DDS\_Sync\_Msg\_Type \*pMsgDump) | |
| **Parameters 목록** | |
| DDS\_Sync\_Msg\_Type \*pMsgDump | 전송할 정보 |
| **Return 값** | |
| DDS\_ReturnCode\_t | DDS manual 참조 |

#### SetMSGWriter

Writer를 초기화하는 함수

|  |  |
| --- | --- |
| **인터페이스** | |
| bool SetMSGWriter(char \*pTopicName, DDSDomainParticipant \*participant) | |
| **Parameters 목록** | |
| char \*pTopicName | Message를 전송할 Topic Name |
| DDSDomainParticipant \*participant | 참여할 도메인 클래스 |
| **Return 값** | |
| bool | true: 성공  false: 실패 |

#### SetMSGReader

Reader를 초기화하는 함수

|  |  |
| --- | --- |
| **인터페이스** | |
| bool SetMSGReader(char \*pTopicName, DDSDomainParticipant \*participant) | |
| **Parameters 목록** | |
| char \*pTopicName | Message가 수신될 Topic Name |
| DDSDomainParticipant \*participant | 참여할 도메인 클래스 |
| **Return 값** | |
| bool | true: 성공  false: 실패 |

#### SetMain

CDDS\_Sync\_Main 을 설정하는 함수

|  |  |
| --- | --- |
| **인터페이스** | |
| void SetMain(CDDS\_Sync\_Main \*pParent) | |
| **Parameters 목록** | |
| CDDS\_Sync\_Main \*pParent | CDDS\_Sync\_MSG를 Member로 가지는 부모클래스 |
| **Return 값** | |
| void |  |

## 고가용 알고리즘 sequence diagram

고가용성 알고리즘은 크게 heartbeat 송수신 부분, 일반적인 DDS 메시지 처리 부분, 장애절체 및 복구 등 3가지 부분으로 나눠질 수 있다.

#### Heartbeat 메시지 송수신 sequence diagram

본 연구에서는 heartbeat 메시지의 송수신을 통하여 상대방이 동작 중임을 확인하는 방법을 사용하고 있다. 이는 본 연구에서뿐만 아니라, 일반적으로 상대 어플리케이션의 동작 유무를 판별하는 매우 일반적인 방법이다.

본 연구에서 Heartbeat 메시지는 active 에서 standby로 전송된다. Standby는 일정 주기 동안 heartbeat 메시지 수신을 기다리며 (이 주기는 config.h 파일에 정의되어 있다), standby 어플리케이션은 정해진 시간 동안 heartbeat 메시지가 수신되지 않으면 active 에 장애가 발생한 것으로 판단한다. Heartbeat 메시지를 수신하면 다음 번 heartbeat 메시지 수신을 위하여 계속하여 timer를 reset 하여 일정 기간 동안의 heartbeat 메시지 수신을 준비한다.

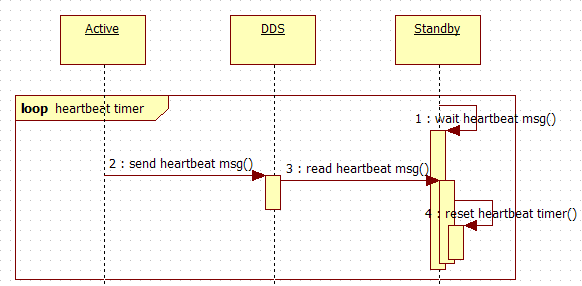


그림 . heartbeat 송수신 sequence diagram

#### 일반적인 전투체계 DDS 메시지(CS DDS) 처리 sequence diagram

그림 11과 같이 Active와 standby는 동시에 CS DDS 메시지를 수신한다. 단지 active 모드의 어플리케이션은 정상적으로 수신된 DDS 메시지를 처리한다. DDS 메시지를 처리한다는 의미는 어플리케이션 본연의 임무를 수행한다는 의미이다. 예를 들면, 트랙 관리 어플리케이션은 source track 메시지(CS DDS 메시지를 의미함)를 수신하여, 항적 융합 또는 시스템 트랙 생성하여 전송(그림 11의 “11 : send processed DDS msg”)하게 된다.

Standby 어플리케이션은 수신된 DDS 메시지를 처리하지 않고 queue에 저장을 한다. 이렇게 queue에 저장된 DDS 메시지는 active 어플리케이션의 처리가 완료되었음을 의미하는 메시지를 수신하는 경우(그림 11의 “9 : send processed CS info” 및 “10 : update CS DDS msg in queue” 메시지) queue에서 삭제된다. queue에서 삭제하는 이유는 추후 재전송되는 경우를 방지하기 위함이다.

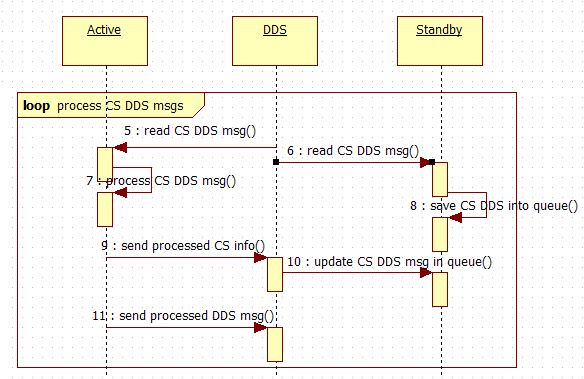


그림 . 전투체계(CS DDS) 메시지 sequence diagram

#### 장애절체 및 복구 sequence diagram

그림 12의 “12 : fault”는 active 어플리케이션에서 장애가 발생했음을 의미하며, “15 : fail to send heartbeat msg” 메시지는 장애가 발생하여 heartbeat 메시지를 전송하지 못함을 의미한다. 또한 “19 : recovery”는 장애로 인하여 정지된 active 어플리케이션의 오류 조건을 제거한 후 재 시작 시키는 행위를 의미한다.

Standby 어플리케이션은 정해진 시간 동안 active 어플리케이션으로부터 heartbeat 메시지 수신을 못하는 경우, active에 장애가 발생한 것으로 판단하여 장애절체 모드에 들어가게 된다. Standby 어플리케이션은 장애절체를 위하여 현재 동작중인 heartbeat timer를 해제하고(“14 : fired heartbeat timer”), 내부 상태를 active 모드로 변경한 후(“16: set active mode”), active에 의해서 처리되지 않은 queue에 저장된 메시지를 처리하여(“17 : process CS DDS msg in queue”) 정송하게 된다(“18 : send processed DDS msg”). 이 이후부터는 그림 11의 절차를 반복하게 된다.

그림 12의 “20 : wait heartbeat msg” 에서 재기동된 active 어플리케이션은 자신의 상태를 결정짓기 위하여 일단 heartbeat 메시지의 수신을 일정 시간 동안 대기한다. 이 경우 이미 동작중인 active 어플리케이션이 있기 때문에 heartbeat 메시지를 수신하게 되며(“21 : read heartbeat msg”), 자신의 모드를 standby 모드로 결정하게 된다(“22 : set standby mode”). 이렇게 standby 모드로 결정된 후에는 현재 동작중인 active 어플리케이션과 내부 상태를 맞추기 위하여 동기화 요청 메시지를 보내게 되며(“23 : request syncing” & “24 : request syncing”), 동기화 요청 메시지를 수신한 active 어플리케이션은 현재 내부 상태를 standby 어플리케이션에 보내게 된다(“25 : send syncing data”). 동기화 데이터를 수신한(“26: read syncing data”) standby 어플리케이션은 프로세스 내부 상태를 active와 동기화 시켜(“27 : update process status”), 향후 수신되는 전투체계 메시지에 대한 처리 준비를 끝마치게 된다.

여기서 장애절체 시 소요되는 시간은 “12 : fault”로인하여 heartbeat 메시지를 보내지 못함으로 인하여 소요되는 시간 ∇Theartbeat\_timer(=heartbeat timer 평균 대기 시간) + queue에 저장된 메시지 처리 시간 ∇Tqueue\_length 으로 정의된다. 이 시간의 특성은 사용자가 설정한 heartbeat 송수신 주기와 내부 CS DDS 메시지 처리 시간에 비례하게 되며, 이는 응용프로그램에 매우 종속적이기 된다. 단지 일반적으로 heartbeat 송수신 주기가 짧을수록 장애절체가 빨라지게 되나, 단점은 잦은 heartbeat 송수신은 시스템의 성능을 떨어뜨리고, 네트워크 회선 사용량이 늘게 되어 전체 네트워크 상태를 떨어뜨릴 수 있기 때문에 각 응용에 맞추어 설정해야 한다. 또한 ∇Tqueue\_length 시간의 특징은 메시지 양(즉, queue의 길이)과 단위 메시지를 처리하는데 소요되는 시간이다. 메시지 처리에 소요되는 시간은 어플리케이션 로직 및 CPU 성능에 비례하고, 이를 상수로 가정하는 경우 queue length가 적정한 길이를 갖도록 하는 설정이 필요하다. 다음에 제시할 예와 같이 일반적으로 queue length 설정은 ∇Theartbeat\_timer 시간과 밀접하다. 예를 들어, 자함 센서가 자함 정보를 10Hz로 송신하는 경우, 장애 판단을 위한 ∇Theartbeat\_timer 은 3초로 설정한 경우 queue length 는 10Hz x 3초 = 30 + 여유율 20%(6개)로 대략 36개 정도의 길이를 갖도록 정의하게 된다.

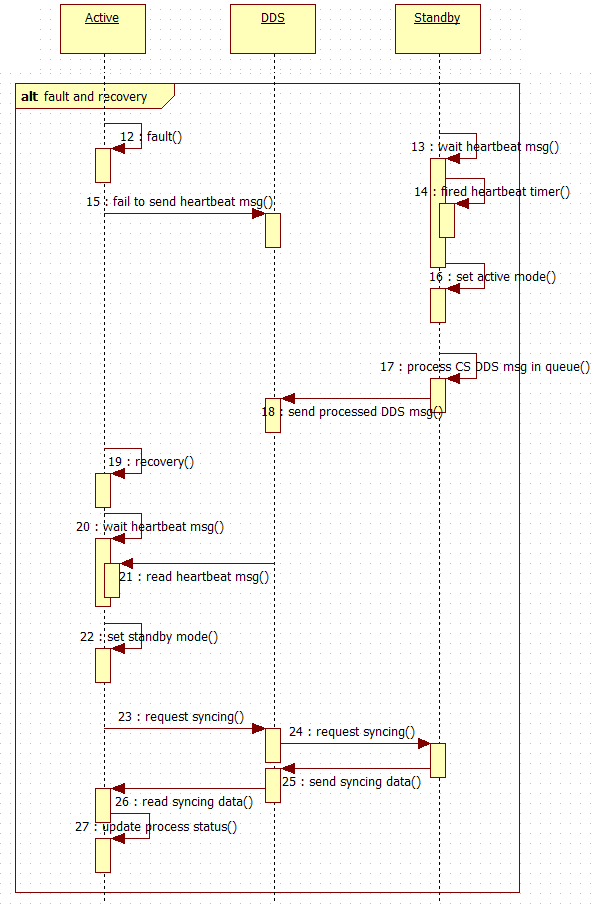


그림 . 장애절체 및 복구 sequence diagram

## 고가용 알고리즘 프로그래밍 가이드

고가용성 알고리즘의 정확한 사용을 위하여 전반적인 프로그래밍 순서에 대하여 설명한다.

1. 헤더 추가

CDDS\_Sync\_Main, CDDS\_Sync\_Log 를 사용하기 위해서 해당 헤더 파일을 Include 한다.

**#include "DDS\_Sync\_Main.h"**

1. Log 사용하기

Log 를 파일을 생성할 때 CDDS\_Sync\_Main::getInstance().OpenLog 를 사용하고, 저장할 땐 CDDS\_Sync\_Log::getInstance().PrintLog 를 사용한다.

Error가 발생 했을 때 CDDS\_Sync\_Main::getInstance().GetLastError 를 사용한다.

int main(int argc, TCHAR\* argv[])

{

:

:

:

**// 저장로그 설정**

if(!**CDDS\_Sync\_Main::getInstance().OpenLog("c:\\DDS\_SYNC\_LOG.txt")**)

{

char cErrMsg[256];

**// 에러 읽기**

**CDDS\_Sync\_Main::getInstance().GetLastError(cErrMsg);**

**// 로그 저장**

**CDDS\_Sync\_Log::getInstance().PrintLog("Warning: fail to open %s log file\n", "c:\\DDS\_SYNC\_LOG.txt");**

}

:

:

:

}

1. 초기화

int main(int argc, TCHAR\* argv[])

{

:

:

:

**// 동기화 시작**

if (**CDDS\_Sync\_Main::getInstance().StartSync(argv[1], argv[2])** == false)

{

CDDS\_Sync\_Log::getInstance().PrintLog("Error: invalid topics(\"%S\", \"%S\") for StartSync\n");

return -1;

}

**// 동기화를 위해 사용 될 콜백 함수들 설정.**

**CDDS\_Sync\_Main::getInstance().SetDataCallback(Callback\_Send, Callback\_Receive);**

:

:

:

}

1. Listener에 Type 추가

사용되는 DDS Type Index를 저장하도록 Listener에 추가한다.

class CDDSListener : public DDSDataReaderListener {

private:

CALLBACK\_FUNC m\_CallBack;

public:

CDDSListener():m\_CallBack(NULL){;}

**int m\_iType;**

**void setType(int iType)**

**{**

**m\_iType = iType;**

**}**

:

:

:

}

1. DDS message receive callback Function 설정

사용자가 설정하는 DDS message receive callback 함수를 CDDS\_Sync\_Main::getInstance().SetCallback를 사용하여 CDDS\_Sync\_Main 내부에서 처리 할 수 있도록 설정한다.

bool CreateReader( std::string strQosName, CALLBACK\_FUNC pCallBackFunc )

{

bool bResult = true;

//CreateDomain( strQosName );

if( CreateTopic() )

{

:

:

if ( m\_pReader == NULL )

bResult = false;

else

{

// Callback 가변으로변경

//m\_cListener.SetCallBackFunc((CALLBACK\_FUNC)pCallBackFunc);

int iType = **CDDS\_Sync\_Main::getInstance().SetCallback((CALLBACK\_FUNC)pCallBackFunc);**

**m\_cListener.setType(iType);**

}

}

return bResult;

}

1. DDS message receive callback Function 처리

사용자가 정의한 DDS message 가 receive 되면 CDDS\_Sync\_Main::getInstance().MessageReceive 를 사용하여 CDDS\_Sync\_Main 내부에서 처리 되도록 수정한다

virtual void on\_data\_available(DDSDataReader\* reader)

{

:

:

:

for (int i = 0; i < tDataSeq.length(); ++i) {

if (tInfoSeq[i].valid\_data) {

**CDDS\_Sync\_Main::getInstance().MessageReceive(m\_iType, &tDataSeq[i], sizeof(tDataSeq[i]), tInfoSeq[i].publication\_sequence\_number, tInfoSeq[i].source\_timestamp);**

// printf("receved message\n");

}

}

:

:

:

}

1. 동기화 데이터 송신

Active 상태에서 Standby가 활성화 되면서 동기화 데이터를 요청하는 경우 CDDS\_Sync\_Main::getInstance().SetSendData를 사용하여 동기화 할 데이터를 송신한다.

static void Callback\_Send(void\*)

{

printf("Callback\_Send\n");

DWORD dwSize = 1024\*1024; // is more than MAX\_TRACK \* sizeof(SystemTrack\_t) + sizeof(int) of \*gpnSysTrack

char \*pSendData = new char[dwSize]; // malloc enough big buffer for gpnSysTrack & gpSysTrack

memcpy(pSendData, gpnSysTrack, sizeof(int));

memcpy(pSendData + sizeof(int), gpSysTrack, sizeof(SystemTrack\_t) \* \*gpnSysTrack);

if( !**CDDS\_Sync\_Main::getInstance().SetSendData(pSendData, sizeof(int) + sizeof(SystemTrack\_t) \* \*gpnSysTrack)** )

{

printf("Error SetSendData\n");

}

delete[] pSendData;

for(int iCnt = 0; iCnt <= \*gpnSysTrack; iCnt++)

{

printf("%d) %d, %d, %d\n",iCnt, gpSysTrack[iCnt].nSysTrackNo, gpSysTrack[iCnt].nX,gpSysTrack[iCnt].nY);

}

}

1. 동기화 데이터 수신

Standby 상태에서 Active가 보낸 동기화 데이터를 받아서 동기화 할 수 있도록 = CDDS\_Sync\_Main::getInstance().GetReceiveDataSize 를 사용하여 동기화 할 데이터 전체 크기를 얻고, CDDS\_Sync\_Main::getInstance().GetReceiveData 통해 동기화 데이터를 수신한다.

static void Callback\_Receive(void\*)

{

printf("Callback\_Receive --> Done !!!\n");

**DWORD dwSize = CDDS\_Sync\_Main::getInstance().GetReceiveDataSize();**

if( dwSize <= 0 )

return;

char \*pSendData = new char[dwSize];

if( !**CDDS\_Sync\_Main::getInstance().GetReceiveData(pSendData)** )

{

printf("Error GetReceiveData\n");

}

memcpy(gpnSysTrack, pSendData, sizeof(int));

memcpy(gpSysTrack, pSendData + sizeof(int), sizeof(SystemTrack\_t) \* \*gpnSysTrack);

for(int iCnt = 0; iCnt <= \*gpnSysTrack; iCnt++)

{

printf("%d) %d, %d, %d\n",iCnt, gpSysTrack[iCnt].nSysTrackNo, gpSysTrack[iCnt].nX,gpSysTrack[iCnt].nY);

}

delete[] pSendData;

}

1. Active, Standby의 확인

CDDS\_Sync\_Main::getInstance().GetMode() 를 이용해 현재 Mode를 확인할 수 있다.

void RecvMsg\_Track(void\* pMsg, DDS\_SequenceNumber\_t MsgNo, DDS\_Time\_t SourceTime)

{

:

:

:

nCount++;

// Active 경우의 처리 로직 (예)

if( **CDDS\_Sync\_Main::getInstance().GetMode() == PLAY\_MODE::DDS\_SYNC\_MODE\_ACTIVE** )

{

if( (nCount%1000) == 0 )

{

printf("Active (%d), DDS sequence no: %d, %d, Sec : %ld, NanoSec : %ld\n",

nCount, MsgNo.high, MsgNo.low, SourceTime.sec, SourceTime.nanosec);

}

}

else if( **CDDS\_Sync\_Main::getInstance().GetMode() == PLAY\_MODE::DDS\_SYNC\_MODE\_STANDBY** )

{

// Standby 경우의 처리 로직 (예)

if( (nCount%1000) == 0 )

{

printf("Standby (%d), DDS sequence no: %d, %d, Sec : %ld, NanoSec : %ld\n",

nCount, MsgNo.high, MsgNo.low, SourceTime.sec, SourceTime.nanosec);

}

}

free(pSysTrack);

}

## Memory Manager Design Specification

### 개요

2차 연구보고서의 작성 이후 사용자의 편리성을 높이기 위하여 추가된 memory manager에 대한 설계 문서는 다음과 같다.

임베디드 시스템을 고려한 Memory Manger는 과거 대부분의 개발 환경이 C 언어에 맞춰져 있기 때문에 C 언어를 지원하도록 개발되었으나, 본 과제를 위하여 C++ 인터페이스를 추가하였다.

### C 인터페이스

Memory Manger 함수는 아래와 같이 크게 3개의 인터페이스를 갖는다.

#### mm\_init() 함수

Parameters: None

Description: 내부적으로 정의된 크기 (default 32767 byte) buffer를 할당하고, 관련된 변수의 초기화를 수행한다

Return: None

#### mm\_malloc() 함수

Parameters: bufsize\_t size

Description: 원하는 크기의 버퍼 크기를 할당하여 리턴한다.

Return: 할당된 버퍼 포인터. 만약 에러인 경우 NULL을 리턴한다.

#### mm\_calloc() 함수

Parameters: bufsize\_t size

Description: mm\_malloc 함수를 이용하여 버퍼를 할당하고, 할당된 영역을 0으로 초기화하여 리턴한다..

Return: 할당된 버퍼 포인터. 만약 에러인 경우 NULL을 리턴한다.

#### mm\_realloc() 함수

Parameters: void \*buffer, bufsize\_t size

Description: mm\_malloc을 통하여 할당된 버퍼 포인터와 새롭게 원하는 버퍼의 크기를 받아 재할당을 하게된다.

Return: 할당된 버퍼 포인터. 만약 에러인 경우 NULL을 리턴한다.

#### mm\_free() 함수

Parameters: void \*buf

Description: 할당된 버퍼를 재할당을 위하여 반납한다.

Return: None

### C++ 인터페이스

C++ 인터페이스는 아래의 class diagram으로 대체한다. 각 인터페이스 함수는 C 인터페이스 함수를 wrapping 하였으므로 C 인터페이스의 각 함수와 동일한 기능을 수행한다.



그림 . 메모리 관리자 class diagram

### 성능 평가

성능 평가는 C/C++ 개발 시 주로 사용하는 standard C library 함수인 malloc/free 함수를 기준으로 비교하였다. 기본적으로 malloc/free 함수는 멀티 쓰레드 환경에서 운영될 수 있도록 구현되어 있다. 때문에 싱글 쓰레드 환경에서는 약간의 오버헤드를 가지게 된다. 본 연구에서 제시된 Memory Manager는 성능의 최적화를 위하여 싱글 쓰레드 환경 및 멀티 쓰레드 환경을 구분하여 성능을 평가 하였다.

성능 평가 방법은 무작위로 malloc/free 함수 및 그에 상응하는 함수를 반복 호출하여 전체 수행 시간을 측정하는 방법을 사용하였다. 그 결과는 아래 표와 같다. 성능 평가는 Windows 7, CPU i5, Memory 4G 환경에서 수행하였다.

테스트 방법은 Memory Manager와 함께 구현된 테스트 프로그램의 TestProg 변수를 조절하여 반복 횟수를 조정하고, MultiThread 값을 0 또는 1로 정의하여 single thread와 multi thread 환경을 테스트하였다. 단, Multi thread 조건인 경우라도 주요 부분의 locking 기능은 사용하되 single thread인 경우와 동일한 테스트 조건을 만족하기 위하여 테스트 쓰레드는 하나만을 수행하였다.

표 . Memory Manager 성능 평가 결과

|  |  |  |  |
| --- | --- | --- | --- |
| 함수 반복횟수 | 10,000 반복 | 20,000 반복 | 40,000 반복 |
| malloc/free | 0.072417 초 | 0.166893 초 | 0.325632 초 |
| Single thread 조건 | 0.041863 초 | 0.082059 초 | 0.172725 초 |
| Multi thread 조건 | 0.044746 초 | 0.095513 초 | 0.188255 초 |

위 표 1에서 보는 바와 같이 standard C malloc/free 함수를 이용하는 경우에 비하여 multithread를 고려한 환경에서도 대략 2배 정도의 성능 향상을 보이고 있음을 확인할 수 있다.

## Track Generator Application Codes

탐색 레이더의 소스 트랙을 생성하는 데모 프로그램은 Visual Studio 2008을 통하여 개발하였으며, 기본적으로 제공되는 document view 구조의 프로그램에 마우스 이벤트 핸들러만을 추가 하였다. 사용자는 마우스의 왼쪽 버튼을 이용하여 두 지점을 선택하면 두 지점을 반복하여 이동하면서 소스 트랙을 생성하여 DDS로 전송한다. 주요 모듈의 코드는 다음과 같다.

|  |
| --- |
| void CTrackGeneratorView::OnDraw(CDC\* pDC)  {  CTrackGeneratorDoc\* pDoc = GetDocument();  ASSERT\_VALID(pDoc);  if (!pDoc)  return;  if (!NULL\_POINT(m\_ptFirst))  {  // PT\_RADIUS simulate the coverage of surveillance radar  pDC->Ellipse(m\_ptFirst.x - PT\_RADIUS, m\_ptFirst.y - PT\_RADIUS, m\_ptFirst.x + PT\_RADIUS, m\_ptFirst.y + PT\_RADIUS);  pDC->MoveTo(m\_ptFirst);  }  if (!NULL\_POINT(m\_ptSecond))  {  pDC->Ellipse(m\_ptSecond.x - PT\_RADIUS, m\_ptSecond.y - PT\_RADIUS, m\_ptSecond.x + PT\_RADIUS, m\_ptSecond.y + PT\_RADIUS);  pDC->LineTo(m\_ptSecond);  // generate source track  TrackGenerator(m\_ptFirst, m\_ptSecond, 0.005, 10, pDC);  }  }  //////////////////////////////////////////////////////////////////////////  // Function: TrackGenerator  // Parameters:  // ptFirst: start point  // ptSecond: end point  // dInterval: the distance of two adjacent points.  // nIntervalSleep: sleep time of two adjacent points that simulate rotation of  // the surveillance radar  // pDC: Windows device context  // Return:  // 0: always  // Description  // divide the line consisting of ptFirst and ptSecond into dInterval of  // two consecutive points using parametric line equation  //////////////////////////////////////////////////////////////////////////  int CTrackGeneratorView::TrackGenerator(CPoint ptFirst, CPoint ptSecond, double dInterval, int nIntervalSleep, CDC\* pDC)  {  CPoint pt;  double dTotal = dInterval;  CBrush \*pbr = pDC->GetCurrentBrush();    pDC->SelectStockObject(HOLLOW\_BRUSH);  while (dTotal <= 1.0)  {  // if start point and end point is NULL, then return  if (NULL\_POINT(m\_ptFirst) && NULL\_POINT(m\_ptSecond))  break;  // parametric equation of line  pt.x = ptFirst.x + dTotal \* (ptSecond.x - ptFirst.x);  pt.y = ptFirst.y + dTotal \* (ptSecond.y - ptFirst.y);  if (m\_nDistanceOfTwoPoint == 0)  {  // calc the distance of first point and pt  m\_nDistanceOfTwoPoint = sqrt(pow(pt.x - m\_ptFirst.x, 2.0) + pow(pt.y - m\_ptFirst.y, 2.0)) \* RADAR\_RATIO;  // draw radar coverage  pDC->Ellipse(ptFirst.x - m\_nDistanceOfTwoPoint, ptFirst.y - m\_nDistanceOfTwoPoint,  ptFirst.x + m\_nDistanceOfTwoPoint, ptFirst.y + m\_nDistanceOfTwoPoint);  TargetTrackGenerator(ptFirst, m\_nDistanceOfTwoPoint, 100, pDC);  }  pDC->Ellipse(pt.x - PT\_RADIUS, pt.y - PT\_RADIUS, pt.x + PT\_RADIUS, pt.y + PT\_RADIUS);  // draw radar coverage  pDC->Ellipse(pt.x - m\_nDistanceOfTwoPoint, pt.y - m\_nDistanceOfTwoPoint,  pt.x + m\_nDistanceOfTwoPoint, pt.y + m\_nDistanceOfTwoPoint);  TargetTrackGenerator(pt, m\_nDistanceOfTwoPoint, 100, pDC);  dTotal += dInterval;  Sleep(nIntervalSleep);  }  pDC->Ellipse(ptSecond.x - m\_nDistanceOfTwoPoint, ptSecond.y - m\_nDistanceOfTwoPoint,  ptSecond.x + m\_nDistanceOfTwoPoint, ptSecond.y + m\_nDistanceOfTwoPoint);  TargetTrackGenerator(ptSecond, m\_nDistanceOfTwoPoint, 100, pDC);  pDC->SelectObject(pbr);  // clear the client area and regenerate the tracks  Invalidate(TRUE);  return 0;  }  //////////////////////////////////////////////////////////////////////////  // Function: TargetTrackGenerator  // Parameters:  // pt: the origin point of circle  // R: radius of circle  // nTrackNum: the number of source track in the circle that will be generated  // Description  // generate nTrackNum source tracks in radius R of point pt  //////////////////////////////////////////////////////////////////////////  int CTrackGeneratorView::TargetTrackGenerator(CPoint pt, double R, int nTrackNum, CDC\* pDC)  {  CPoint ptTrack = pt;  srand((unsigned)time(NULL));  MSG\_TR\_SOURCE\_RADAR tMsg;  for(int i = 0; i < nTrackNum; i++)  {  double dRandVal = GEN\_RAND;  ptTrack.x = pt.x + R \* GEN\_RAND \* cos (dRandVal \* 2 \* M\_PI);  ptTrack.y = pt.y + R \* GEN\_RAND \* sin (dRandVal \* 2 \* M\_PI);  // send source track to DDS  memset( &tMsg, 0, sizeof(MSG\_TR\_SOURCE\_RADAR) );  tMsg.source\_track\_number = i;  tMsg.position\_x = ptTrack.x;  tMsg.position\_y = ptTrack.y;  LDDS\_WRITE( 1, MSG\_TR\_SOURCE\_RADAR, &tMsg );  // display one source track to screen  pDC->Ellipse(ptTrack.x -1, ptTrack.y - 1, ptTrack.x + 1, ptTrack.y + 1);  }  return 0;  } |

## Track Management Application Codes

아래와 같이 source track으로부터 간단한 알고리즘을 통하여 system track을 구성하여 active 모드인 경우엔 DDS로 전송하고, standby 모드인 경우엔 장애를 대비하여 대략 3초간의 저장 능력을 가진 circular queue에 저장하여 recovery 시의 무결성에 대비하기 위한 코드는 아래와 같다

|  |
| --- |
| //////////////////////////////////////////////////////////////////////////  // Function: svc  // Description:  // This function read source track message from Message Queue and  // make system track, and send the system track  //////////////////////////////////////////////////////////////////////////  virtual int svc()  {  while(1)  {  ACE\_Message\_Block \*mb = 0;  SystemTrack\_t \*pSysTrk = NULL;  // read source track message from queue  if (this->getq(mb) == -1)  break;  if (mb->msg\_type() == ACE\_Message\_Block::MB\_HANGUP)  {  mb->release();  break;  }  else  {  MSG\_TR\_TARGET\_INFO MsgTargetInfo;  // process the message  pSysTrk = (SystemTrack\_t\*)mb->rd\_ptr();  // make system track  MakeSystemTrack(pSysTrk);  memset(&MsgTargetInfo, 0, sizeof(MSG\_TR\_TARGET\_INFO));  MsgTargetInfo.position\_x = (DDS\_Float)pSysTrk->nX;  MsgTargetInfo.position\_y = (DDS\_Float)pSysTrk->nY;    if (gbActive)  {  // active mode  LDDS\_WRITE( 1, MSG\_TR\_TARGET\_INFO, &MsgTargetInfo);  }  else  {  // standby mode  // put MsgTargetInfo into Queue for failover  SaveMsgToCircularQueue(&MsgTargetInfo);  }  if (pSysTrk->nSysTrackNo == 99)  {  // surveillance radar complete one rotation  \*gpnSysTrack = 0;  }  mb->rd\_ptr(sizeof(SystemTrack\_t));  mb->release();  }  }  ACE\_Reactor::instance()->end\_event\_loop();  return 0;  }  //////////////////////////////////////////////////////////////////////////  // Function: RecvTargetInfo  // Parameters:  // pMsg: system track message that is sended by active RecvDDS\_Track  // MsgNo: system track generation number that is managed by DDS library  // Description:  // receive system track message from active in order to save this message  // to circular queue that is used to recovery of lost message in failover  //////////////////////////////////////////////////////////////////////////  void RecvTargetInfo(void \*pMsg, DDS\_SequenceNumber\_t MsgNo/\*DDS\_Time\_t SourceTime\*/)  {  static int i = 1;  MSG\_TR\_TARGET\_INFO \*pMsg1 = (MSG\_TR\_TARGET\_INFO\*)pMsg;  if(gbActive == true)  printf("Info: Active: RecvTargetInfo: time: %d, %d, Recved count: %d\n", MsgNo.high, MsgNo.low/\*SourceTime.sec, SourceTime.nanosec\*/, i++);  else  {  printf("Info: Standby: RecvTargetInfo: time: %d.%d, Recved count: %d\n", MsgNo.high, MsgNo.low/\*SourceTime.sec, SourceTime.nanosec\*/, i++);  // save message to circular queue  SaveToCircularQueue(pMsg, MsgNo);  }  }  //////////////////////////////////////////////////////////////////////////  // Function: RecvMsg\_Track  // Parameters:  // pMsg: source track that is from surveillance radar  // MsgNo: DDS publisher generate this sequence no on each message  // Description:  // - DDS listener callback of source track  //////////////////////////////////////////////////////////////////////////  void RecvMsg\_Track(void\* pMsg, DDS\_SequenceNumber\_t MsgNo/\*DDS\_Time\_t SourceTime\*/)  {  static unsigned int nCount = 1;  SystemTrack\_t \*pSysTrack = (SystemTrack\_t\*)calloc(1, sizeof(SystemTrack\_t));  if (pSysTrack == NULL)  {  printf("Error: fail to malloc\n");  return;  }  MSG\_TR\_SOURCE\_RADAR\* p\_tMsg = (MSG\_TR\_SOURCE\_RADAR\*)pMsg;  // convert MSG\_TR\_SOURCE\_RADAR to SystemTrack\_t  ConvertRdrMsgToSysTrack(p\_tMsg, pSysTrack);  ACE\_Message\_Block \*mb = new ACE\_Message\_Block(sizeof SystemTrack\_t);  mb->copy((const char\*)pSysTrack, sizeof(SystemTrack\_t));  gSysTrack.putq(mb);  printf("DDS CB: positionX : %.2f, positionY : %.2f\t(%d), DDS sequence no: %d, %d\n",  p\_tMsg->position\_x, p\_tMsg->position\_y, ++nCount, MsgNo.high, MsgNo.low/\*SourceTime.sec, SourceTime.nanosec\*/);  free(pSysTrack);  }  //////////////////////////////////////////////////////////////////////////  // Function: Callback\_Send  // Parameters:  // None  // Description:  // This callback is called when there is need to send shared variables if fail occur  // This callback is called in active mode.  //////////////////////////////////////////////////////////////////////////  static void Callback\_Send(void\*)  {  printf("Callback\_Send\n");  DWORD dwSize = 1024\*1024; // is more than MAX\_TRACK \* sizeof(SystemTrack\_t) + sizeof(int) of \*gpnSysTrack  char \*pSendData = new char[dwSize]; // malloc enough big buffer for gpnSysTrack & gpSysTrack  memcpy(pSendData, gpnSysTrack, sizeof(int));  memcpy(pSendData + sizeof(int), gpSysTrack, sizeof(SystemTrack\_t) \* \*gpnSysTrack);  if( !CDDS\_Sync\_Main::getInstance().SetSendData(pSendData, sizeof(int) + sizeof(SystemTrack\_t) \* \*gpnSysTrack) )  {  printf("Error SetSendData\n");  }  delete[] pSendData;  }  //////////////////////////////////////////////////////////////////////////  // Function: Callback\_Receive  // Parameters:  // None  // Description:  // This callback is called to receive shared variables if fail occur  // This callback is called in standby mode.  //////////////////////////////////////////////////////////////////////////  static void Callback\_Receive(void\*)  {  DWORD dwSize = CDDS\_Sync\_Main::getInstance().GetReceiveDataSize();    if( dwSize <= 0 )  return;  char \*pSendData = new char[dwSize];  if( CDDS\_Sync\_Main::getInstance().GetReceiveData(pSendData) )  {  printf("Error GetReceiveData\n");  }  memcpy(gpnSysTrack, pSendData, sizeof(int));  memcpy(gpSysTrack, pSendData + sizeof(int), sizeof(SystemTrack\_t) \* \*gpnSysTrack);  delete[] pSendData;  }  // <<<---  int main(int argc, TCHAR\* argv[])  {  mm\_init();  // active & standby DDS message test  if (argc > 1 && strcmp(argv[1], "active") == 0)  gbActive = true;  else  gbActive = false;  // initialize the synchronization mechanism  if(!CDDS\_Sync\_Main::getInstance().OpenLog("c:\\DDS\_SYNC\_LOG.txt"))  {  CDDS\_Sync\_Main::getInstance().GetLastError(&uErrID, cErrMsg);  }  CDDS\_Sync\_Main::getInstance().StartSync(argv[1], argv[2]);  CDDS\_Sync\_Main::getInstance().SetDataCallback(Callback\_Send, Callback\_Receive);  //////////////////////////////////////////////////////////////////////////  // initialize the share variables  //////////////////////////////////////////////////////////////////////////  gpSysTrack = (SystemTrack\_t\*)bget(sizeof(SystemTrack\_t) \* MAX\_TRACK); // 16 \* 500 = 80,000 bytes  gpnSysTrack = (int \*)bget(sizeof(int));  \*gpnSysTrack = 0;  gSysTrack.activate();  LDDS\_CREATE\_TOPIC(1, MSG\_TR\_SOURCE\_RADAR, "Reliable", false, true, &RecvMsg\_Track);  if (gbActive)  // In active mode, create MSG\_TR\_TARGET\_INFO writer, But not receiver  LDDS\_CREATE\_TOPIC(1, MSG\_TR\_TARGET\_INFO, "Reliable", true, false, NULL);  else  // In standby mode, create MSG\_TR\_TARGET\_INFO writer and receiver  LDDS\_CREATE\_TOPIC(1, MSG\_TR\_TARGET\_INFO, "Reliable", true, true, &RecvTargetInfo);  // reactor event loop  ACE\_Reactor::instance()->run\_reactor\_event\_loop();  brel(gpSysTrack);  brel(gpnSysTrack);  return 0;  } |

1. Standby 어플리케이션은 장애 복구를 위하여 수신된 메시지를 일정 기간동안 내부 queue에 보관하며, active 어플리케이션에서 전송하는 처리 완료 메시지를 수신하여 queue 상태를 갱신하게 된다. 자세한 사항은 “장애절체 및 복구 sequence diagram” 절차를 참조한다. [↑](#footnote-ref-1)