共有メモリ型データポート接続機能

2015/12/22 09:45 - n-ando

ステータス:	終了	開始日:	2015/12/22
優先度:	通常	期日:	2016/03/25
担当者:	miyamoto	進捗率:	100%
カテゴリ:		予定工数:	30.00時間
対象バージョン:	RELEASE_1_2_0		
説明			
同一ノード内のコンポーネント間のデータポート接続において、マーシャリング後のデータのPutを共有メモリ経由で行うことで、TCP/IPスタックを経由することなく高速にデータ転送する機能を実装すること。			

関係しているリビジョン

リビジョン 641 - 2016/02/01 14:24 - miyamoto

[incompat,new_func,new_file,->RELENG_1_2] communication method using shared memory has been implemente. refs #3410

リビジョン 652 - 2016/02/01 20:17 - miyamoto

[incompat,new_func,new_file,->RELENG_1_2] communication method using shared memory has been implemente. refs #3410

リビジョン 660 - 2016/02/25 23:22 - miyamoto

[incompat,new_func,new_file,->RELENG_1_2] add SharedMemory.idl and SharedMemory.py. refs #3410

リビジョン 670 - 2016/02/27 10:36 - miyamoto

[incompat,bugfix,func,->RELENG_1_2] bug fix. refs #3410

リビジョン 671 - 2016/02/27 10:52 - miyamoto

[incompat,bugfix,func,->RELENG_1_2] bug fix. refs #3410

リビジョン 673 - 2016/02/27 11:35 - miyamoto

[incompat,bugfix,func,->RELENG_1_2] bug fix. refs #3410

リビジョン 677 - 2016/02/28 00:58 - miyamoto

[compat,bugfix,->RELENG_1_2] bug fix. refs #3410

リビジョン 679 - 2016/02/29 12:21 - miyamoto

[compat,bugfix,->RELENG_1_2] bug fix. refs #3410

リビジョン 680 - 2016/02/29 22:11 - miyamoto

[compat,bugfix,->RELENG_1_2] bug fix. refs #3410

リビジョン 687 - 2016/03/07 10:21 - miyamoto

[compat,bugfix,->RELENG_1_2] bug fix. refs #3410

リビジョン 695 - 2016/03/12 13:10 - miyamoto

[compat,bugfix,->RELENG_1_2] bug fix. refs #3410

リビジョン 699 - 2016/03/16 16:45 - miyamoto

[compat,bugfix,->RELENG_1_2] bug fix. refs #3410

リビジョン 725 - 2016/10/20 14:42 - miyamoto

[incompat,new_func,->RELENG_1_2] refs #3410

リビジョン 726 - 2016/10/20 19:05 - miyamoto

[compat,bugfix,->RELENG_1_2] fixed SharedMemory.py. refs #3410

リビジョン 727 - 2016/10/20 19:37 - miyamoto

[compat,bugfix,->RELENG_1_2] fixed SharedMemory.py. refs #3410

履歴

#1 - 2016/01/14 16:21 - miyamoto

- 期日を2016/03/25にセット
- 担当者をmiyamotoにセット
- 対象バージョンをRELEASE_1_2_0にセット
- 進捗率を0から50に変更
- 予定工数を30.00時間にセット

#2 - 2016/01/14 23:26 - miyamoto

- ファイル test_SharedMemory.py を追加

実装

Push型

以下のクラスを実装した。

InPortSHMProvider

インポートプロバイダクラス

InPortCorbaCdrProviderクラスを継承している。
InterfaceTypeは「shared_memory」に設定する。

InPortSHMConsumer

インポートコンシューマクラス

InPortCorbaCdrConsumerクラスを継承している。

FactoryInit関数でInPortSHMProviderInit関数とInPortSHMConsumerInit関数を呼び出すことでファクトリが登録され「shared_memory」のInterfaceTypeで通信可能になる。

通信までの手順を説明する。

InPortSHMProviderのコンストラクタ

```
self.shm_address = str(OpenRTM_aist.uuid1())
self._properties.append(OpenRTM_aist.NVUtil.newNV("dataport.shared_memory.address",self.shm_address))
UUIDにより共有メモリの空間名を生成し、dataport.shared_memory.addressというプロパティに保存する。
```

InPortSHMConsumerクラスのinit関数

```
self.shm_address = prop.getProperty("shared_memory.address")
if self.shm_address:
    if self._shmem is None:
        self._shmem = mmap.mmap(0, 256, self.shm_address, mmap.ACCESS_WRITE)
```

プロパティから取得した空間名から共有メモリを生成する。

init関数はInPortBaseクラスのcreateConsumer関数とInPortPushConnectorクラスのコンストラクタの2回呼び出されるが、createConsumer関数で呼び出された時には空間名のプロパティがないため共有メモリを生成するのはInPortPushConnectorクラスのコンストラクタとなっている。現在は動作確認のためにデータサイズを256に固定しているが、送信するデータのサイズによって変更する必要がある。

InPortSHMConsumerクラスのput関数

```
self._shmem.seek(os.SEEK_SET)
self._shmem.write(data)
data_size = len(data)
mar_data_size = cdrMarshal(CORBA.TC_ushort, data_size)
return self.convertReturnCode(inportcdr.put(mar_data_size))
```

共有メモリにデータを書き込み、CORBAで符号化したデータサイズだけ送信している。この方法とは別にデータサイズも共有メモリに書き込むという方法も考えられる。

InPortSHMProviderクラスのput関数

```
data_size = cdrUnmarshal(CORBA.TC_ushort, data)
self._shmem = mmap.mmap(0, data_size, self.shm_address, mmap.ACCESS_READ)
shm_data = self._shmem.read(data_size)
ret = self._connector.write(shm_data)
return self.convertReturn(ret, shm_data)
```

データサイズを復号化し、共有メモリからデータを取り出す。

Pull型

以下のクラスを実装した。

OutPortSHMProvider

アウトポートプロバイダクラス

OutPortCorbaCdrProviderクラスを継承している。

OutterfaceTypeは「shared_memory」に設定する。

OutPortSHMConsumer

アウトポートコンシューマークラス

OutPortCorbaCdrConsumerクラスを継承している。

Push型と同じくOutPortSHMProviderInit関数とOutPortSHMConsumerInit関数を呼び出すことでファクトリを登録する。

通信の手順を説明する。

OutPortSHMProviderクラスのコンストラクタ

```
self.shm_address = str(OpenRTM_aist.uid1())
```

```
self._properties.append(OpenRTM_aist.NVUtil.newNV("dataport.shared_memory.address",self.shm_address))
```

```
self._shmem = mmap.mmap(0, 256, self.shm_address, mmap.ACCESS_WRITE)
```

空間名の生成、プロパティへの保存、共有メモリの生成を行う。

OutPortSHMConsumerクラスのinit関数

```
self.shm_address = prop.getProperty("shared_memory.address")
```

空間名を取得する。

現在のところ何故かOutPortPullConnectorのコンストラクタではOutPortConsumerオブジェクトのinit関数が呼ばれないため、実験のためにinit関数が呼ばれるようにしてある。

OutPortSHMProviderクラスのget関数

```
self._shmem.seek(os.SEEK_SET)
```

```
self._shmem.write(cdr[0])
```

```
data_size = len(cdr[0])
```

```
mar_data_size = cdrMarshal(CORBA.TC_ushort, data_size)
```

```
return self.convertReturn(ret, mar_data_size)
```

共有メモリにデータを書き込み符号化したデータサイズを送信する。

OutPortSHMConsumerクラスのget関数

```
data_size = cdrUnmarshal(CORBA.TC_ushort, cdr_data)
```

```
self._shmem = mmap.mmap(0, data_size, self.shm_address, mmap.ACCESS_READ)
```

```
shm_data = self._shmem.read(data_size)
```

```
data[0] = shm_data
```

データサイズを復号化して共有メモリから読み込んだデータを書き込む。

テスト

添付したテスト用コードでテストを行った。

テスト用コードは以下の動作を行う。

setUp関数

マネージャ初期化

```
self.manager = OpenRTM_aist.Manager.init(sys.argv)
```

```
self.manager.activateManager()
```

データポート生成

```
self._d_in = RTC.TimedLong(RTC.Time(0,0),0)
```

```
self._inIn = OpenRTM_aist.InPort("in", self._d_in)
```

```
prop = OpenRTM_aist.Properties()
```

```
self._inIn.init(prop)
```

```
self.inport_obj = self._inIn.getPortRef()
```

インポートとアウトポートを生成

test_Push関数

データが正常に通信できているかの確認

```
self._d_out.data = 100
```

```
self._outOut.write()
```

```
ret = self._inIn.isNew()
```

```
self.assertTrue(ret)
```

```
data = self._inln.read()
self.assertEqual(data.data, 100)
```

test_Pull関数

データが正常に通信できているかの確認

```
self._d_out.data = 100
self._outOut.write()
data = self._inln.read()
self.assertEqual(data.data, 100)
```

#3 - 2016/02/18 14:32 - n-ando

rtc.confで設定できるようにする

```
port.outport.shem_default_size: <byte>
1M: byte 1M
2k: byte 2k
1024: 1024 byte
port.inport.shem_default_size: 1M
```

最初は小さいようようで、変化したら再確保

ロックの方法について調査

#4 - 2016/02/24 00:20 - miyamoto

ロックの方法について

共有メモリでミューテックスを共有する方法

Linuxの場合は共有メモリでミューテックスを共有できる。

```
int fd = shm_open(vecfile, O_RDWR|O_CREAT, S_IRWXU);
char* ptr = (char*)mmap(NULL, 1024, PROT_READ|PROT_WRITE, MAP_SHARED, vecfd, 0);
close( fd );
pthread_mutex_t* mutex = (pthread_mutex_t*)&ptr[0];
*mutex = PTHREAD_MUTEX_INITIALIZER;
pthread_mutexattr_t attr;
pthread_mutexattr_init(&attr);
pthread_mutexattr_setpshared(&attr, PTHREAD_PROCESS_SHARED);
pthread_mutex_init(mutex, &attr);
```

Windowsでの方法はまだ調査中

その他の方法

現在の実装ではInPortSHMConsumerのput関数で、

- 共有メモリにデータ書き込み
- InPortSHMProviderのput関数呼び出し リモート呼び出ししたput関数内で共有メモリからデータ読み込み

という手順を踏んでおり、さらに空間名はUUIDによって決めているためInPortSHMConsumerのput関数内をロックしてしまえば、RTCが複数の実行コンテキストで駆動している場合でも問題は起きないと考えられます。

具体的には以下のように共有メモリ書き込みからput関数リモート呼び出しまでをロックする。

```
guard = OpenRTM_aist.ScopedLock(self._mutex)
self._shmem.seek(os.SEEK_SET)
self._shmem.write(data)
data_size = len(data)
mar_data_size = cdrMarshal(CORBA.TC_ulong, data_size)
ret = inportcdr.put(mar_data_size)
del guard
```

共有メモリの実装について

Linuxでshm_open関数を使用した場合、/dev/shm以下に一時ファイルが作成される。
このファイルの内容をmmap関数で仮想アドレスにマッピングする事で共有メモリが使用できる。

Windowsの場合、同様の機能は存在しないみたいです。

ReadProcessMemory、WriteProcessMemory関数を使用する事で他のプロセスのメモリ領域を操作することは一応できます。

WindowsでCreateFileMapping関数やMapViewOfFile関数を使用して共有メモリを実装する例がありますが、これはハードディスクに作成したファイルの内容を仮想アドレスにマッピングする関数です。
CreateFileMapping関数でファイルハンドルを渡さなかった場合、ページングファイルに領域を確保します。
Linuxで言えば、open関数を使用してファイルを開いてmmap関数でマッピングする処理と同じです。

WindowsでPythonのmmapモジュールを使用した場合、CreateFileMapping関数を使用した場合と同じくハードディスクにファイルを作成して仮想アドレスにマッピングするという処理をしています。
現在の実装ではmmapモジュールを使用しているため、ページングファイル上に領域を作成してマッピングしています。

#5 - 2016/02/25 23:14 - miyamoto

- ファイル test_SharedMemory.py を追加

共有メモリ通信のためのインターフェースとしてSharedMemory.idlを作成した。

```
interface SharedMemory
{
    void create_memory(in long memory_size, in string shm_address);
    void open_memory(in long memory_size, in string shm_address);
    void close_memory(in boolean unlink);
    void setInterface(in SharedMemory sm);
    PortStatus put();
    PortStatus get();
};
```

create_memory

初期化を行う。

Windowsではページングファイル上に領域を確保して、Linuxでは/dev/shm/以下にファイルを作成する。作成したファイルの内容をmmapにより仮想アドレスにマッピングする。

open_memory

既に確保済みの共有メモリ領域を仮想アドレスにマッピングする。

close_memory

アンマップ、共有領域の削除を行う。

unlinkをTrueにした場合は/dev/shm/以下に作成したファイルを削除する。

setInterface

通信先のCORBAインターフェースを設定する。

設定することによりcreate_memoryを呼び出したときに、通信先での共有メモリ領域のマッピングを自動的に行う。また設定したメモリのサイズをデータのサイズが上回った場合に、通信先の初期化も自動的に行う。

put

送信を知らせる。

putを呼び出す前に共有メモリにデータを書き込んでおくことにより、送信先でデータを読み込む事ができる。

get

送信を要求する。

getの処理内で共有メモリへのデータの書き込みを行う事により、送信元でデータを読み込む事ができる。

また共有メモリ操作クラスとしてSharedMemory.pyを作成した。

SharedMemoryクラスは上記のインターフェースによる操作ができるようになっている。

SharedMemoryクラスには以下の関数を定義してある。

string_to_MemorySize

1M、1k等のデータのサイズを表す文字列を数値に変換する

write

データを書き込む

データサイズを先頭8byteに保持させており、通信するデータはその後ろに書き込む。

read

データを読み込む。
先頭8byteからデータサイズを読み込み、取得したサイズだけデータを読み込む。

またInPortSHMProviderはInPortProvider、SharedMemoryを継承するように変更した。
OutPortSHMProviderはOutPortProvider、SharedMemoryを継承するように変更した。

以前はプロパティのdataport.shared_memory.addressで共有メモリ空間名を保持していたが、SharedMemoryインターフェースで渡せるようになったため削除した。

また共有メモリの初期の大きさはOutPort側でcreate_memory関数を呼び出すとInPort側のマッピングも自動的に行われるようになったため、rtc.confのport.outport.out.shem_default_sizeで設定するだけで済むようになった。具体的には以下のように指定する。

```
port.outport.out.shem_default_size: 10M
port.outport.out.shem_default_size: 100k
port.outport.out.shem_default_size: 300
```

#6 - 2016/02/27 01:14 - miyamoto

名前が紛らわしいため、SharedMemory.idlのSharedMemoryインターフェースをPortSharedMemoryに変更した。

```
interface PortSharedMemory
{
    void open_memory(in long memory_size, in string shm_address);
    void create_memory(in long memory_size, in string shm_address);
    void close_memory(in boolean unlink);
    void setInterface(in PortSharedMemory sm);
    PortStatus put();
    PortStatus get();
};
```

#7 - 2016/03/17 11:02 - miyamoto

- 進捗率を50から100に変更

#8 - 2017/08/30 14:18 - n-ando

- ステータスを新規から終了に変更

ファイル

test_SharedMemory.py	2 KB	2016/01/14	miyamoto
test_SharedMemory.py	2.52 KB	2016/02/25	miyamoto