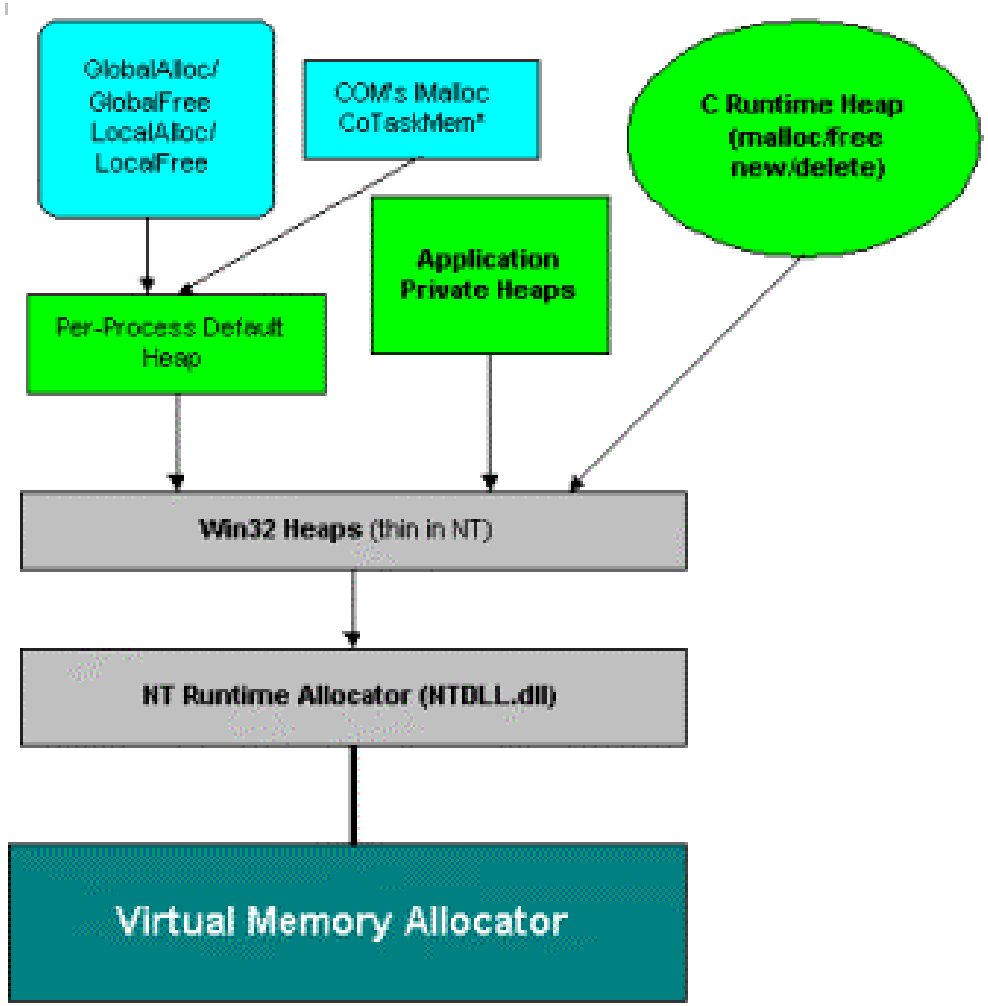
Zarządzanie pamięcią

Zarządzanie pamięcią

System COM, aby mógł śledzić zmiany w alokacji pamięcią podczas wywołań, obsługuje pamięć niezależnie od *Runtime Library* (RT) C++ i korzysta ze sterty bieżącego procesu.



Zarządzanie pamięcią

Do obsługi pamięci COM, zdefiniowany został interfejs `IMalloc`:

```
interface IMalloc : IUnknown
{
  void *Alloc([in] ULONG cb);
  void *Realloc ([in] void *pv, [in] ULONG cb);
  void Free([in] void *pv);
  ULONG GetSize([in] void *pv);
  int DidAlloc(void *pv);
  void HeapMinimize(void);
}
```

Zarządzanie pamięcią

Zawiera on analogiczne metody do funkcji `malloc()`, `realloc()`, `free()`. Dodatkowo dostępne są metody GetSize() zwracająca rzeczywiście zaalokowany rozmiar pamięci, który jest nie mniejszy niż zarezerwowany metodą `Alloc()` lub `Realloc()`.
Metoda `DidAlloc()` pozwala na sprawdzenie, czy wskazany obszar pamięci został zarezerwowany za pomocą metod interfejsu `IMalloc`.

Metoda `HeapMinimize()` minimalizuje obszar sterty systemu COM, uwalniając dla systemu nieużywaną pamięć.

Zarządzanie pamięcią

Wskaźnik do interfejsu IMalloc można uzyskać za pomocą funkcji `CoGetMalloc()`:

```
IMalloc* pMalloc;
CoGetMalloc(1, &pMalloc);
```

System udostępnia również funkcje `CoTaskMemAlloc()`, `CoTaskMemRealoc()` i `CoTaskMemFree()`, które odwołują się do interfejsu `IMalloc`.

Zarządzanie pamięcią

Podczas przekazywania parametrów do metod należy pamiętać, że w przypadku, gdy podczas wywołania spodziewamy się zmiany alokacji obszaru pamięci dla podanego argumentu następującego po różnych stronach interfejsu, to powinien on być zawsze alokowany przy użyciu interfejsu `IAlloc`:

- Parametr typu `[in]` jest alokowany i zwalniany przez klienta. Podsystem COM nie będzie próbował zmieniać alokacji pamięci, tak więc parametr tego typu może być rezerwowany dowolną metodą (`new`, `malloc`, `IMalloc::Alloc()`). Należy pamiętać, że metoda zwalniająca pamięć powinna odpowiadać tej, która ją rezerwowała (odpowiednio `delete`, `free`, `IMalloc::Free()`).

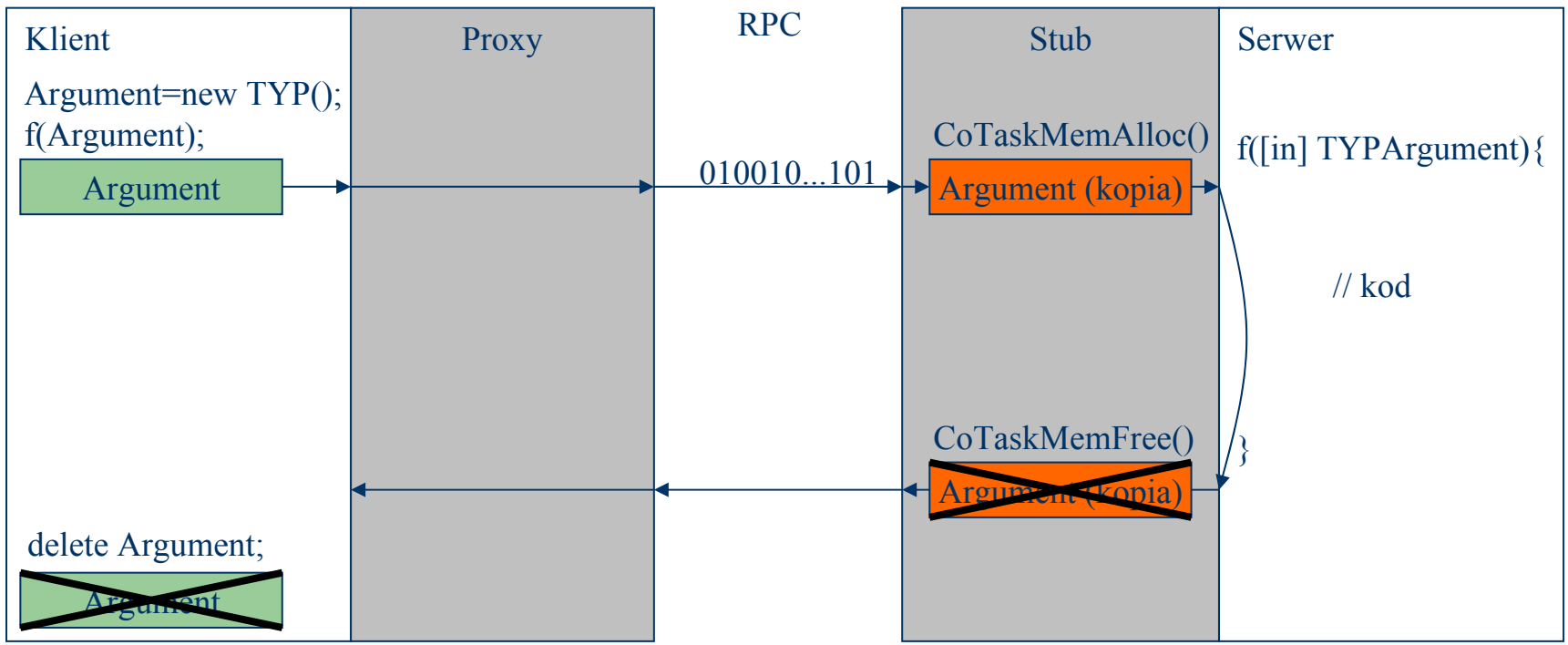
Zarządzanie pamięcią

- Dla parametru typu `[out]` przekazywany jest wskaźnik do istniejącej zmiennej, w której przekazany zostanie wynik operacji. W przypadku danych o zmiennym rozmiarze (łańcuchy tekstowe, wektory), odpowiedni obszar pamięci alokowany jest po stronie komponentu i zwalniany przez klienta. Do alokacji pamięci należy użyć alokatora `IMalloc::Alloc()` i zwalniając pamięć `IMalloc::Free()`.

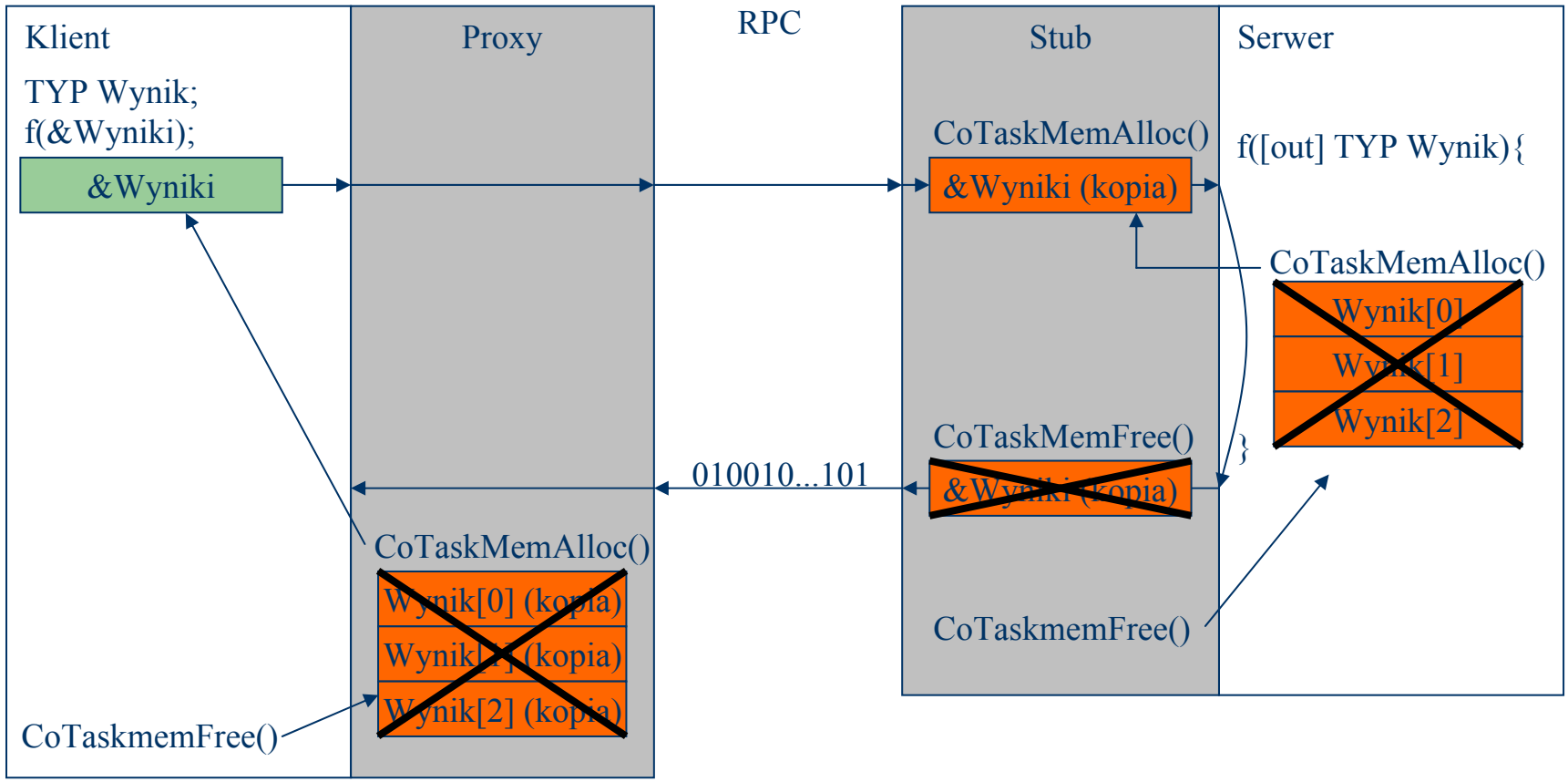
Zarządzanie pamięcią

- Dla parametru typu `[in,out]` przekazywany jest wskaźnik do istniejącej zmiennej, która przechowuje argument dla komponentu. W przypadku danych o zmiennym rozmiarze (łańcuchy tekstowe, wektory) komponent może próbować przealokować lub zwolnić pamięć. Ostatecznie pamięć zwalniana jest przez klienta. Ze względu na możliwość zmiany alokacji pamięci przez komponent konieczne jest użycie alokatora `IMalloc::Alloc()` i zwalnianie pamięci za pomocą `IMalloc::Free()`.

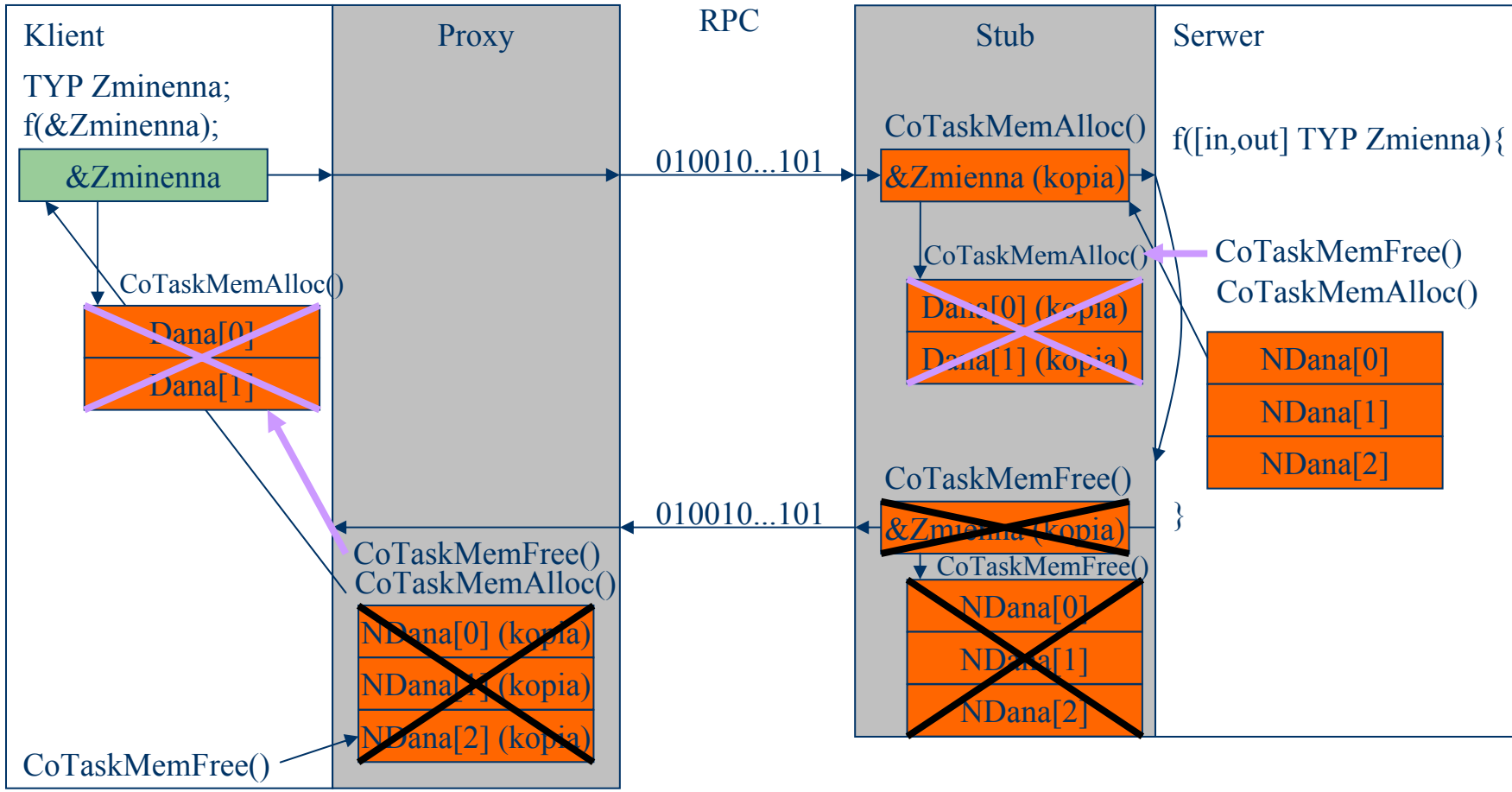
Zarządzanie pamięcią – parametr [in]



Zarządzanie pamięcią – parametr [out]



Zarządzanie pamięcią – parametr [in,out]



Zarządzanie pamięcią

Do alokacji pamięci typu `BSTR` używane są metody `SysAllocString()`, `SysReAllocStringLen()` i `SysFreeString()`. Wewnętrznie używają one interfejsu `IMalloc`.

Zarządzanie pamięcią
Przykład 1

```
interface IStringManipulator : IUnknown
{
        HRESULT SetString([in, string] LPCSTR pString);
        HRESULT SwapString([in, out, string] LPSTR* ppString);
        HRESULT GetString([out, string] LPSTR* ppString);
}
```

Zarządzanie pamięcią
Przykład 1

Komponent:

```
LPSTR myString=NULL

HRESULT CStringManipulator::SetString(LPCSTR pString) {
  myString=(LPSTR)CoTaskMemRealloc(myString, strlen(pString)+1);
  strcpy(myString, pString);
}

HRESULT CStringManipulator:: SwapString(LPSTR* ppString) {
  LPSTR tmpStr=*ppString;
  *ppString=myString;
  myString=tmpStr;
}

HRESULT CStringManipulator::GetString(LPSTR* ppString) {
  *ppString=(LPSTR)CoTaskMemAlloc(strlen(myString)+1);
  strcpy(*ppString, myString);
}
```

Zarządzanie pamięcią
Przykład 1

Klient:

```
LPCSTR String1="Ala ma kota";
LPCSTR String2="Kot ma Ale";
pComponent->SetString(String1);

LPSTR UpdateString=(LPSTR)CoTaskMemAlloc(strlen(String2)+1);
memcpy(UpdateString, String2)
pComponent->SwapString(&UpdateString);
...
CoTaskMemFree(UpdateString);

LPSTR ReceiveString=NULL;
pComponent->GetString(&ReceiveString);
...
CoTaskMemFree(ReceiveString);
```

Zarządzanie pamięcią
Przykład 2

```
interface IStringArrayManipulator : IUnknown
{
  typedef struct {
    int Cout;
    [size_is(Count)] LPSTR* Strings;
  } StringArray;

  HRESULT SetStrings([in] StringArray StrArr);
  HRESULT SwapStrings([in, out] StringArray *pStrArr);
  HRESULT GetStrings([out] StringArray *pStrArr);
}
```

Zarządzanie pamięcią
Przykład 2

Komponent:

```
StringArray myStrArr;
myStrArr.Count=0;
myStrArr.Strings=NULL;

HRESULT CStringArrayManipulator::SetString(StringArray StrArr)
{
  for (int i=0;i<myStrArr.Count;i++)
    CoTaskMemFree(myStrArr.Strings[i]);

  myStrArr.Strings=(LPSTR*)CoTaskMemRealloc(StrArr.Count*sizeof(LPSTR));

  myStrArr.Count=StrArr.Count;
  for (int i=0;i<myStrArr.Count;i++){
    myStrArr.Strings[i]=(LPSTR)CoTaskMemAlloc(strlen(StrArr.Strings[i])+1);
    strcpy(myStrArr.Strings[i], StrArr.Strings[i]);
  }
}
```

Zarządzanie pamięcią
Przykład 2

Komponent:

```
HRESULT CStringArrayManipulator:: SwapStrings(StringArray *pStrArr)
{
  StringArray tmpStrArr=*pStrArr;
  *pStrArr=myStrArr;
  myStrArr=tmpStrArr;
}

HRESULT CStringArrayManipulator::GetStrings(StringArray *pStrArr)
{
  (*pStrArr).Strings=(LPSTR*)CoTaskMemAlloc(myStrArr.Count*sizeof(LPSTR));
  (*pStrArr).Count=myStrArr.Count;
  for (int i=0;i<myStrArr.Count;i++){
    (*pStrArr).Strings[i]=(LPSTR)CoTaskMemAlloc((*pStrArr).Strings[i],
                                          strlen(myStrArr.Strings[i])+1);
    strcpy((*pStrArr).Strings[i], myStrArr.Strings[i]);
  }
}
```