Remedy IT
Your challenge - our solution

Asynchronous Method Invocation through Connectors

IDL to C++11

Why AMI4CCM?

- LwCCM has no standardized way to perform an asynchronous method invocation between two components
- Each project and developer solves this challenge differently
- Need a standardized solution to align implementations and models

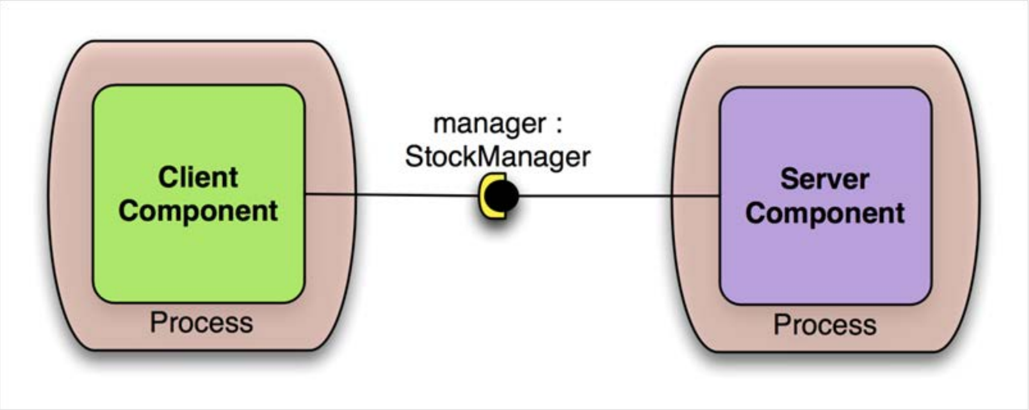
Remedy IT

Your challenge - our solution

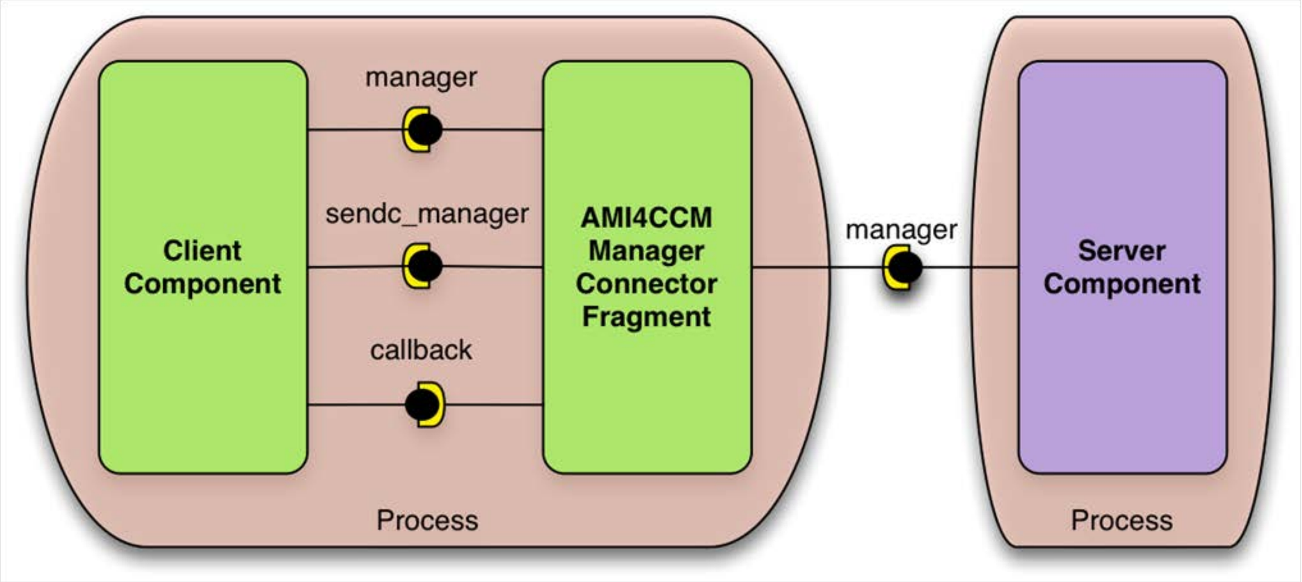
Requirements AMI4CCM

- Pure client side solution
- Shall not extend the LwCCM specification
- Reuse the Generic Interaction Support (GIS) that was introduced by DDS4CCM and which is now part of the CCM specification

AMI4CCM Connector (1)



AMI4CCM Connector (2)



Conversion rules

- AMI4CCM defines a set of conversion rules how to convert a regular interface to
  - Asynchronous invocation interface
  - Reply handler interface
  - Callback for the successful case
  - Callback for the error case
- Naming rules are based on CORBA AMI
  - Reply handler is a local interface

Conclusion

- AMI4CCM introduces asynchronous method invocation for CCM
- Doesn't extend the CCM core
- Reuses GIS from CCM

Remedy IT

Your challenge - our solution

IDL to C++11

Why a new language mapping?

Remedy IT

Your challenge - our solution

- IDL to C++ language mapping is impossible to change because
  - Multiple implementations are on the market (open source and commercial)
  - A huge amount of applications have been developed
- An updated IDL to C++ language mapping would force vendors and users to update their products
- The standardization of a new C++ revision in 2011 (ISO/IEC 14882:2011, called C++11) gives the opportunity to define a new language mapping
  - C++11 features are not backward compatible with C++03 or C++99
  - A new C++11 mapping leaves the existing mapping intact

Remedy IT

Your challenge - our solution

Goals

- Simplify mapping for C++
- Make use of the new C++11 features to
  - Reduce amount of application code
  - Reduce amount of possible errors made
  - Gain runtime performance
  - Speedup development and testing
  - Faster time to market
  - Reduced costs
  - Reduced training time

Remedy IT
Your challenge - our solution

Basic types

IDL	C++11	Default value
short	int16_t	0
long	int32_t	0
long long	int64_t	0
unsigned short	uint16_t	0
unsigned long	uint32_t	0
unsigned long long	uint64_t	0
float	float	0.0
double	double	0.0
long double	long double	0.0
char	char	0
wchar	wchar_t	0
boolean	bool	false
octet	uint8_t	0

String types

No need to introduce an IDL specific type mapping but leverage STL

IDL	C++11
`string name;`	`std::string name;`
`wstring w_name;`	`std::wstring w_name;`
	`name = “Hello”;`
	`std::cout << name << std::endl;`

Sequence

Remedy IT

Your challenge - our solution

IDL unbounded sequence maps to `std::vector`

IDL

```
typedef sequence<long> LongSeq;
```

C++11

```
typedef std::vector <int32_t>
    LongSeq;

LongSeq mysequence;

// Add an element to the vector
mysequence.push_back (5);

// Dump using C++11 range based loop
for (const int32_t& e : mysequence)
{
  std::cout << e << ";" << std::end;
}
```

Reference types (1)

- An IDL interface maps to so called reference types
- Reference types are reference counted, given type A
  - Strong reference type behaves like std::shared_ptr and is available as IDL::traits<A>::ref_type
  - Weak reference type behaves like std::weak_ptr and is available as IDL::traits<A>::weak_ref_type
- A nil reference type is represented as nullptr
- Invoking an operation on a nil reference results in a INV_OBJREF exception

Reference types (2)

Remedy IT

Your challenge - our solution

Given IDL type A the mapping delivers IDL::traits<A> with type traits

IDL

```
interface A
{
  // definitions
};
```

C++11

```
// Obtain a reference
IDL::traits<A>::ref_type a = // obtain a
                           // reference
// Obtain a weak reference
IDL::traits<A>::weak_ref_type w =
    a.weak_reference();

// Obtain a strong reference from a weak
// one
IDL::traits<A>::ref_type p = w.lock ();

if (a == nullptr) // Legal comparisons
if (a != nullptr ) // legal comparison
if (a) // legal usage, true if a != nullptr
if (!a) // legal usage, true if a ==
        // nullptr
if (a == 0) // illegal, results in a
            // compile error
delete a; //  illegal, results in a compile
          //error
```

Components with C++11

Remedy IT

Your challenge - our solution

- Component executors are local object implementations
- Implemented as C++ classes using standardized C++ traits
- Method signatures will be different compared to the IDL to C++ mapping
- Implementation can use the new C++11 features

Remedy IT
Your challenge - our solution

Conclusion IDL to C++11

- IDL to C++11 simplifies programming
- The combination of reference counting and C++11 move semantics make the code much safer and secure
- Application code is much smaller and easier to read

Remedy IT
Your challenge - our solution

CCM Implementations

CIAO as implementation

Remedy IT

Your challenge - our solution

- Open source LwCCM implementation using IDL to C++
  - Optional support for CCM navigation
  - Profile to disable CCM event support
- Uses TAO for all of its infrastructure
- Support for DDS4CCM using OpenDDS and RTI Connext DDS as underlying DDS implementations
- Support for AMI4CCM
- Available from http://download.dre.vanderbilt.edu

CIAOX11 as implementation

- LwCCM implementation using IDL to C++11
- Deviates from LwCCM towards UCM
- Clean separation between CORBA and the user component executors
- Current plan is to release CIAOX11 as open source LwCCM implementation using a similar license as CIAO
- New CIAOX11 high level architecture would also work with other programming languages

Remedy IT
Your challenge - our solution

DAnCE

- Open source implementation of the OMG D&C standard using the IDL to C++ language mapping
- Uses TAO as CORBA implementation
- Introduces LocalityManager as component server
  - CIAOX11 will deliver its own C++11 LocalityManager and will reuse the other services from the C++ DAnCE version
- Defines a set of plugins to customize its behavior