CMPI

The Common Manageability Programming Interface

Konrad Rzeszutek
Viktor Mihajlovski

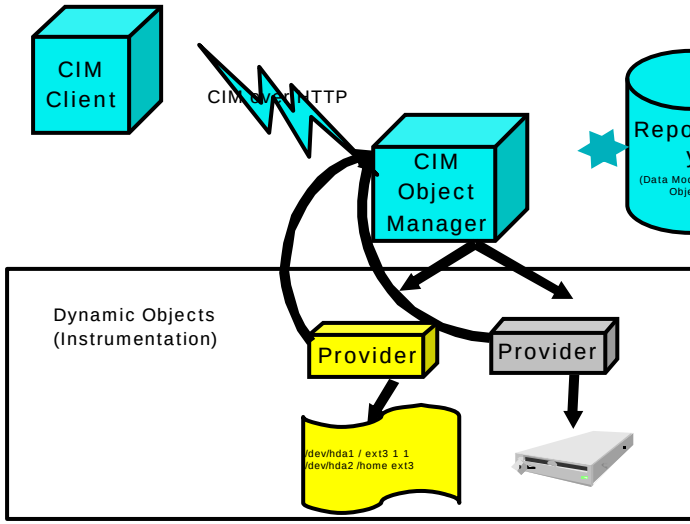
IBM Linux Technology Center

Agenda

- **Motivation: CIMOM Provider Interfaces**
- **CMPI Design Principles**
- **Technology**
- **Availability**
- **Usage**
- **C++ Binding**
- **Remote CMPI**

Motivation: CIMOM Provider Interfaces

- **CIM Object Managers use providers to access and manipulate System Management resources**
- **Providers must follow the implementation rules defined by the CIMOM (programming language, structure)**
- **At the moment multiple CIMOMs are available, all with their own, incompatible Provider Interface**
- **Provider Writers are forced to either select a particular CIMOM or write multiple providers for the same resource**
- **CMPI allows to write providers running under any CIMOM**



CMPI Design Principles

- **The Common Manageability Programming Interface was designed with the following principles in mind**
  - **Facilitation: reduces effort to write a (CIM) provider, i.e. CMPI memory management relieves provider from book-keeping**
  - **Interoperability: allows a provider (even binary) to be deployed to each CIMOM (supporting/supported by CMPI)**
  - **Independence: doesn't require link libraries of any kind**
  - **Completeness: supports all common CIM provider functions as found in the various CIMOMs**
  - **Encapsulation: the provider is not concerned with CIMOM details regarding data type implementation**

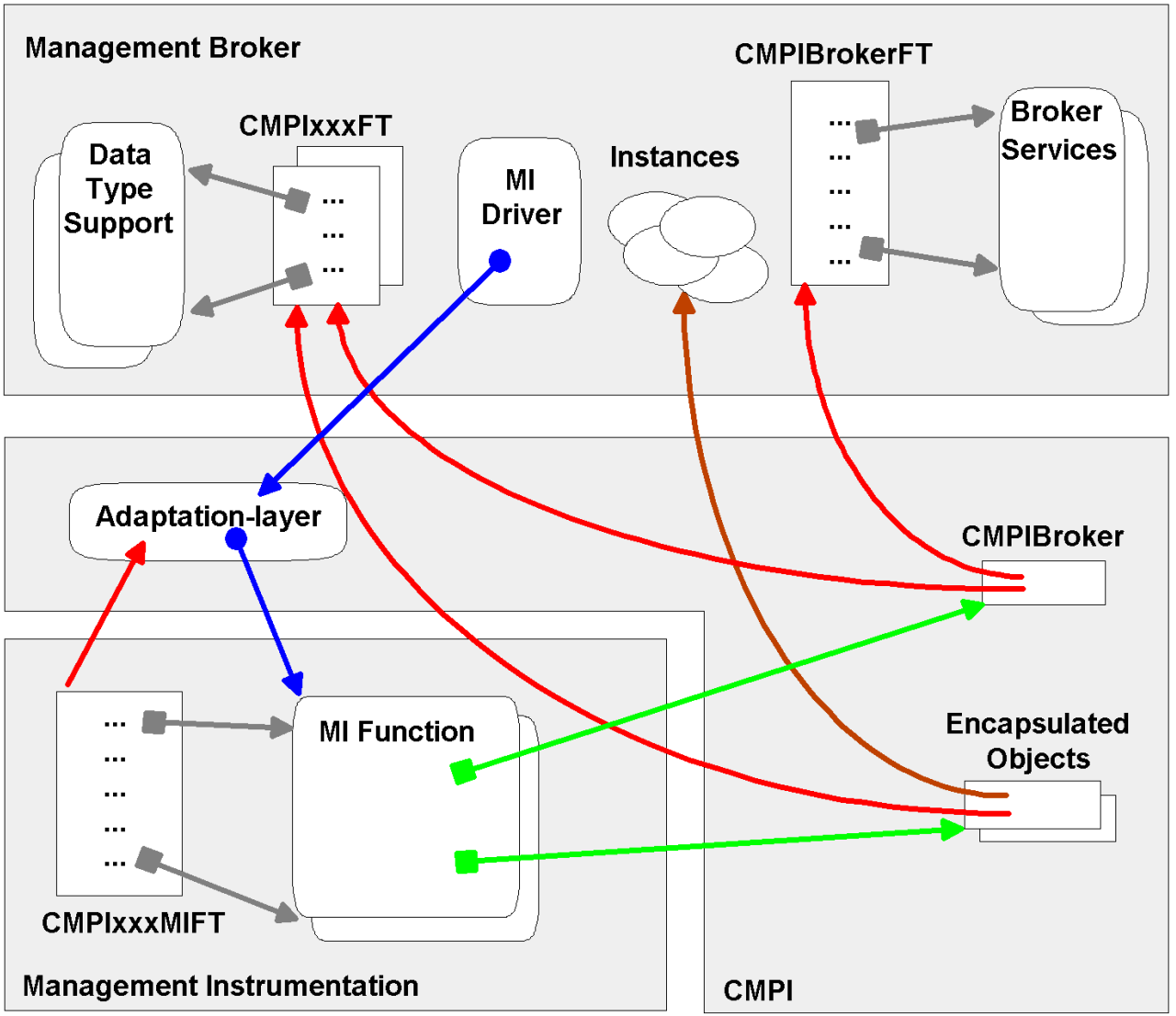
CMPI Rationale ...

- **Scalability: CMPI is thread-safe (reentrant) allowing to perform many provider invocations in parallel if needed**
- **Remoteness: CMPI providers can easily be deployed in a remote environment using Remote CMPI**

CMPI Technology

- **The primary CMPI binding is in ANSI C**
  - **better binary compatibility than C++**
  - **lean implementation**
  - **but offers C++ "convenience interface"**
- **CIMOM/Provider calls via function tables**
  - **includes up- and downcalls and data manipulation**
  - **no need for link libraries**
  - **easily extensible (by adding function pointers at the table's end)**
- **Data Types**
  - **Simple, not managed (int, float, boolean)**
  - **Encapsulated (String, Instance, Property)**
  - **Array (of simple and encapsulated types)**

CMPI Technology...



Availability

- **CMPI Spec hosted by The Open Group, available via wbemSource**
  http://www.wbemsource.org
- **CMPI Implementations**
  - **SNIA/openCIMOM: SBLIM CMPI Adapter**
  - **Pegasus: SBLIM CMPI Adapter, native support in 2.3**
  - **OpenWBEM: native support as beta version**

Usage

- **Determine Instrumentation Type**
  - Instance, Association, Method, Property, Indication
- **Provide a MI Function Table per MI Type**
  - either write a factory function by hand, name format <providername>_Create<mytype>MI, e.g. Printer_CreateInstanceMI
  - use the convenience macro CM<mytype>MIStub, e.g. CMInstanceMIStub("Printer",...)
- **Implement the MI functions pointed to by Function Table**
  - e.g. InstanceMIFT: enumerateInstances, enumerateNames, getInstance, setInstance, createInstance, deleteInstance, execQuery
- **Consider using the SBLIM provider skeleton generator**
  - http://www-124.ibm.com/developerworks/oss/cvs/sblim/psg/

Usage Sample

```
/* instance MI: enumerate instance names
*/

CMPIStatus PrinterEnumInstanceNames(CMPIInstanceMI *mi,
  CMPIContext *ctx, CMPIResult *result, CMPIObjectPath *
{
  CMPIStatus      status = {CMPI_RC_OK,NULL};
  CMPIObjectPath *returnedPath;
  returnedPath = Printer_makePath(op);
  if (returnedPath == NULL) {
    status.rc = CMPI_RC_ERR_FAILED;
  } else {
    CMReturnObjectPath(result,returnedPath);
  }
  CMReturnDone(result);
  return status;
}
```

Usage sample ...

```
/*
* Printer_makePath C version
*/

CMPIObjectPath * Printer_makePath(CMPIObjectPath *ref)
{
   CMPIObjectPath *op = CMNewObjectPath(_broker, NULL,
  "MY_Printer",NULL);

   if (op!=NULL) {
      CMSetNameSpaceFromObjectPath(op,ref);
      CMAddKey(op,"Name",CMPI_chars,"/dev/lpt0");
      ...
   }
   return op;
}
```

C++ Binding

- **C Interface pros**
  - **binary compatibility**
  - **small footprint providers**
- **C Interface cons**
  - **coding is tedious (function pointers, macros)**
  - **error handling**
- **C++ Convenience Support**
  - **Wrappers the C-Interface**
  - **Preserves binary compatibility**
  - **Enhances usability: C++ classes and instances, error handling via exceptions**

Usage Sample (C++)

```
// instance MI: enumerate instance names

CmpiStatus Printer::enumInstanceNames(
  CmpiContext &ctx, CmpiResult &result, CmpiObjectPath &op)
{
  CmpiStatus     status(CMPI_RC_OK);
  CmpiObjectPath returnedPath;
  try {
    returnedPath = Printer_makePath(op);
    result.returnData(returnedPath);
  } catch (CmpiStatus excStatus)
    status = excStatus;
  }
  result.returnDone();
  return status;
}
```

Usage Sample C++ ...

```
/*
* Printer_makePath C++ version
*/

CmpiObjectPath Printer_makePath(CMPIObjectPath &ref)
{
   CMPIObjectPath op(ref.gteNameSpace(),"MY_Printer");

   if (op==NULL)
      throw CMPIStatus(CMPI_RC_ERR_FAILED,"Could not create
  object path ");

   op.setKey("Name",CmpiData("/dev/lpt0"));
      ...

   return op;
}
```

Remote CMPI

- **Extension to allow remote execution of CMPI providers**
  - **Available in 9/2003 via SBLIM project**
- **No changes needed**
  - **The same binaries can be run locally and remotely**
  - **CMPI layer doesn't need to be modified, as the remote CMPI layer behaves like a regular CMPI provider**
- **Efficient**
  - **Small footprint on systems not supporting a full-blown CIMOM (door locks).**
  - **Communication between remote provider and CIMOM only for up-calls and final result transfer**