November 14-17, 2011

MANAGEMENT DEVELOPERS CONFERENCE

San Mateo Marriott
San Mateo, CA

Open Pegasus CIM Server

Part 2 – Advanced Topics

Karl Schopmeyer
Project Coordinator, Pegasus Open Source Project
k.schopmeyer@opengroup.org

This presentation will be available on the MDC and OpenPegasus web sites.

THE Open GROUP
Making standards work®

V1.1 11/17/2011

MANAGEMENT DEVELOPERS CONFERENCE

Agenda

- **Part 1**
  - 1. What is OpenPegasus?
  - 2. What's New?
  - 3. Pegasus Features Overview
  - 4. Technical Subjects
  - 5. How to use and work with Pegasus
  - 6. Issues
  - 7. Discussion and Feedback

- **Part 2 –Advanced Topics**
  - The Pull Operations
  - Registering Pegasus Providers
  - CIM_Error
  - Performance and Resource Utilization
  - Debugging in the Pegasus Environment

If you have subjects for the advanced topics we can try to get them on the list.

Part 2.1
Pull Operations
DMTF DSP0200 V 1.3

Goals Of the Pull Operations

- Parallel existing Enumeration Operations
- Remove deprecated functionality for new operations
- Minimize gratuitous differences
  - Ex. CIMObject vs. CIMInstance (returns instances)
- Create Pull operations for the Enumerates that cause scalability problems
  - Ignore Class and qualifier operations
- Grow error status definitions from existing error status codes
- Keep single error per response philosophy
  - A pull can return data OR an error status code
- Make new things optional

MANAGEMENT DEVELOPERS CONFERENCE

Concept Extensions

- Client sets max size of any pull (maxObjCount)
- Server returns that or fewer objects in response
- Client can terminate response before enumeration exhausted (close operation)
- Filters allow server to filter out things not interesting to client (Filter Language Not Defined)
  - Smarter processing, smaller responses
- Client can ask how big is this response (optional)
  - Return is an estimate not exact count
- Client makes decision on whether error terminates enumeration. (ContinueOnError)

The New Operations

- Open Enumeration Operations
  - OpenEnumerateInstances (EnumerateInstances)
  - OpenEnumerateInstancePaths (EnumerateInstanceNames)
  - OpenReferenceInstances (References)
  - OpenReferenceInstancePaths (ReferenceNames)
  - OpenAssociatorInstances (Associators)
  - OpenAssociatorInstancePaths (AssociatorNames)
  - OpenQueryInstances
- Pull Operations
  - PullInstancesWithPaths
  - PullInstancePaths
  - PullInstances
- Other Operations
  - CloseEnumeration
  - EnumerationCount

Example: OpenEnumerateInstances Operation

- Open new Enumeration. Response is the enumerationContext if the server accepts the open.
- Properties parallel existing operations.
- New properties to allow filtering of objects by server
- Timeout defines time server MUST keep operation open between pulls
- ContinueOnError tells server whether to continue if any pull gets error

OpenEnumerateInstances

<instanceWithPath>> **OpenEnumerateInstances** (
[OUT] <enumerationContext> EnumerationContext
[OUT] Boolean EndOfSequence
[IN] <className> ClassName,
[IN,OPTIONAL] boolean DeepInheritance = true,
[IN,OPTIONAL] boolean IncludeClassOrigin = false,
[IN,OPTIONAL,NULL] string PropertyList [] = NULL,
[IN,OPTIONAL,NULL] string FilterQueryLanguage = NULL,
[IN,OPTIONAL,NULL] string FilterQuery = NULL,
[IN,OPTIONAL] uint32 OperationTimeout,
[IN,OPTIONAL] Boolean ContinueOnError = false,
[IN, OPTIONAL] uint32 MaxObjectCount = 0)

Common Parameters for Opens

- **IN Parameters**
  - FilterQueryLanguage
    - Future – Not yet defined
  - FilterQuery
    - Future
  - OperationTimeout
    - IntraOperation Timeout. Set by client and modifiable by server. Sets min time server must keep context open between operations
  - ContinueOnError
    - Client requests server to continue returning objects despite errors.
  - MaxObjectCount
    - Max count of objects server is to return on this operation. Server may modify this downward. NOTE: 0 is legal.
- Out Parameters
  - EndOfSequence
    - Signals that server has completed operations
  - EnumerationContext
    - Returned by server. Must be supplied with pull and close operations

Pull Example

- Pull a defined number of instances for the defined enumerationContext.
- Server may return up to the defined number of objects
- Server indicates enumeration exhausted with EndOfSequence parameter
- Server returns 0 or more response objects or error status with:
  - EnumerationContext and EndOfSequence indicator

PullInstancesWithPath

```
<instanceWithPath>* PullInstancesWithPath (
      [IN,OUT] <enumerationContext> EnumerationContext,
      [IN] uint32 MaxObjectCount,
      [OUT] Boolean EndOfSequence
     )
```

Close Operation

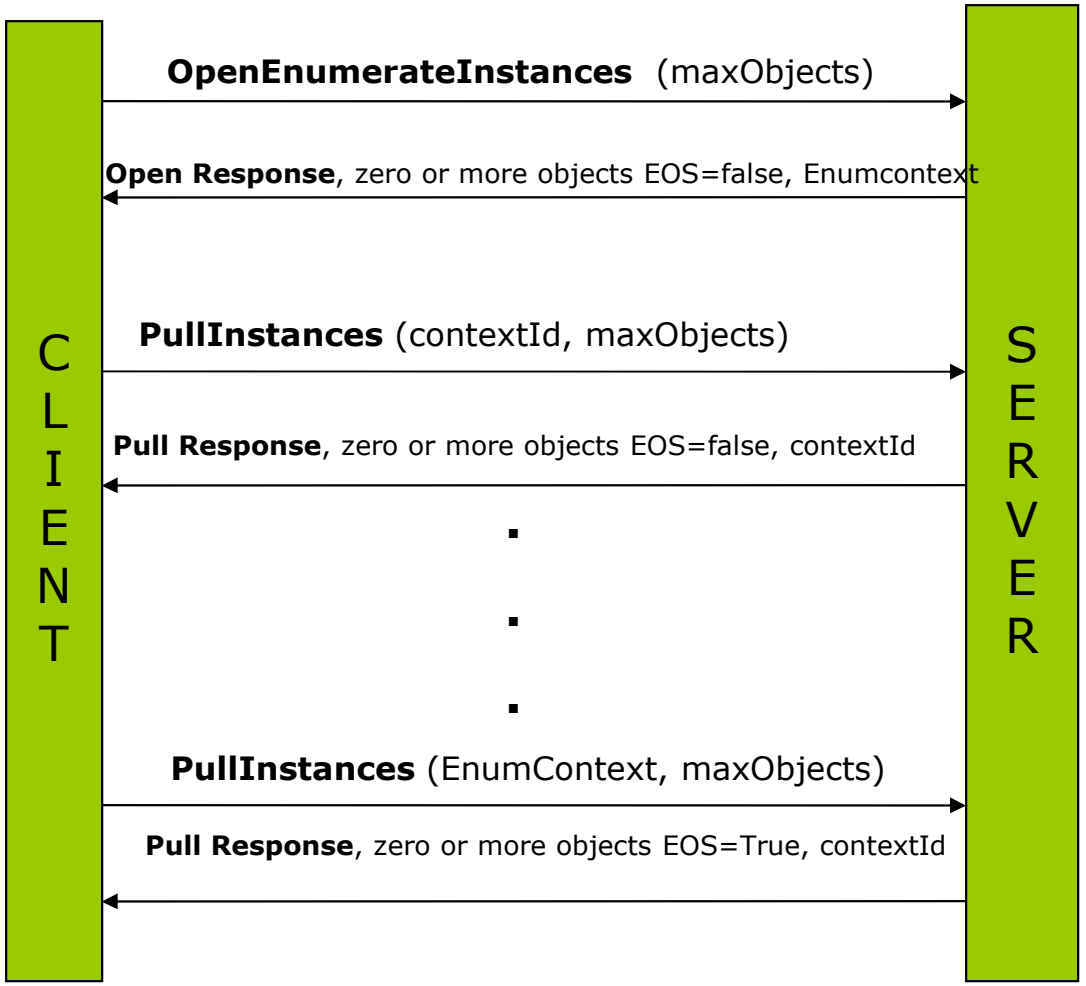
Requests server to close an existing enumeration before the enumeration is exhausted

CloseEnumeration

```
void CloseEnumeration (
        [IN] <enumerationContext> EnumerationContext
)
```

Example: Instance Pull sequence

- **Client opens with OpenEnumerateIndstances**
- **Server responds**
  - EndOfSequence = false
  - Zero or more objects up to count defined by maxObjects in request
  - EnumerationContext
- **Client requests more instances (CIMPullEnumerate)**
  - Enumeration context from open
  - MaxObjects
- **Server Responds**
  - EndOfSequence = false
  - More objects
  - EnumerationContext
- **Client Pull request**
- **Server Responds**
  - EOS=True (indicates no more objects after this response
  - Zero or more objects



Whats New In OpenPegasus

- Client API level
  - Extend client for new pull operations
    - New APIs correspond to CIM operations
- Extend Core server to handle new operations
- NOTE: Works with existing Providers
- Iterator class for response processing

New Pegasus Client Operations

- Open
  - Array<CIMInstance> openEnumerateInstances
  - Array<CIMObjectPath> openEnumerateInstancePaths
  - Array<CIMInstance> openReferenceInstances
  - Array<CIMObjectPath> openReferenceInstancePaths
  - Array<CIMInstance> openAssociatorInstances
  - Array<CIMObjectPath> openAssociatorInstancePaths
- Pull
  - Array<CIMInstance> pullInstancesWithPath
  - Array<CIMObjectPath> pullInstancePaths
- Close
  - void closeEnumeration
- Misc
  - Uint64Arg enumerationCount

Inter OperationTimeouts

- Specified by client as part of open
- May be adjusted downward by server
- Represents minimum time server will keep context open between client calls. Time from end of previous operation to start of next operation.
- Each client call for a context restarts this timer.
- Client may update this without getting objects by requesting 0 entities in request.
- const Uint32Arg& maxObjectCount = Uint32Arg(0)

maxObjectCount
Number of Objects Requested

- Client defines maximum number of objects client wants
- Set on each operation (open and pull)
- NULL value not defined.
- Optional – If not provided, server set size.
- Server responses with the maxObjectCount or fewer objects
- Client may request 0. Server returns no objects but restarts the interOperation timer.

MANAGEMENT DEVELOPERS CONFERENCE

Differences

- Incorporate new parameters
  - maxObjectCount
  - Filter properties
  - Operation Timeout
  - OperationContext
- New Client types
  - Uint32Arg – Allows handling Uint32 with NULL.
  - OperationContext – new Class that provide opaque handling of Client receive and send of the OperationContext parameter

Overview of Pegasus CIM Operation Responses

- Provider interfaces multithreaded
  - Each CIM operation request gets its own thread
- Operation responses are incremental
  - Provider can deliver one or more objects with each call to deliver response objects.
- Responses are queued through serverand aggregated for needs of delivery
- Provider delivery thread blocked to support delivery.

Conclusion: Pegasus was largely ready to handle pull operations without provider changes.

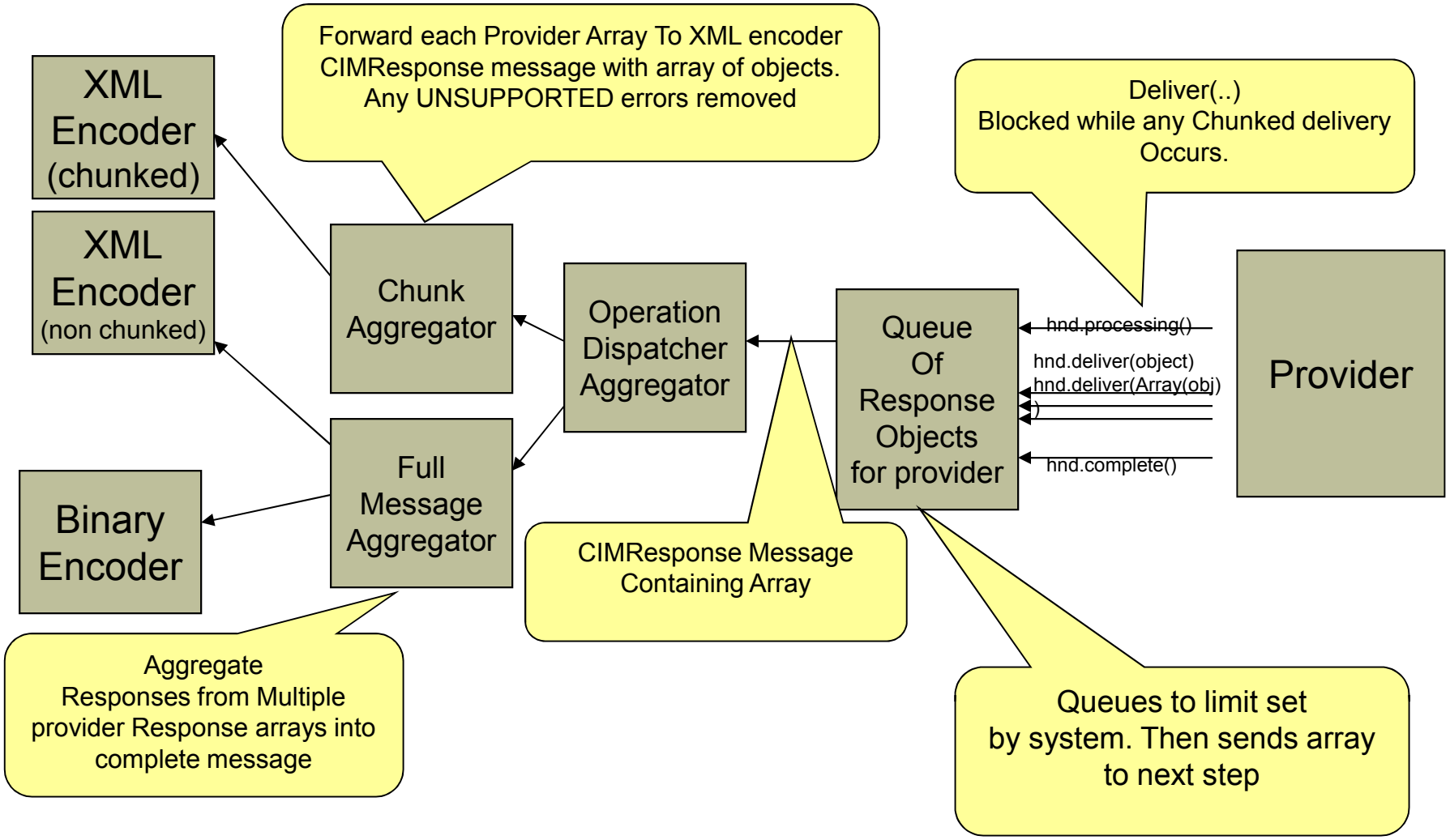
Pegasus Provider Response Interface

- Each CIM Operation request type defines specific handlers for responses
- Each CIM Operation call provides handler ref
- Each call gets Response Handler object
  - Response calls are:
    - hnd.processing() – start response
    - hnd.deliver(…) – deliver one or more response entities (CIMInstances, CIMObjects, CIMObjectpaths)
      - deliver() interfaces have both single object and array definition.
    - hnd.complete() – provider finished delivering

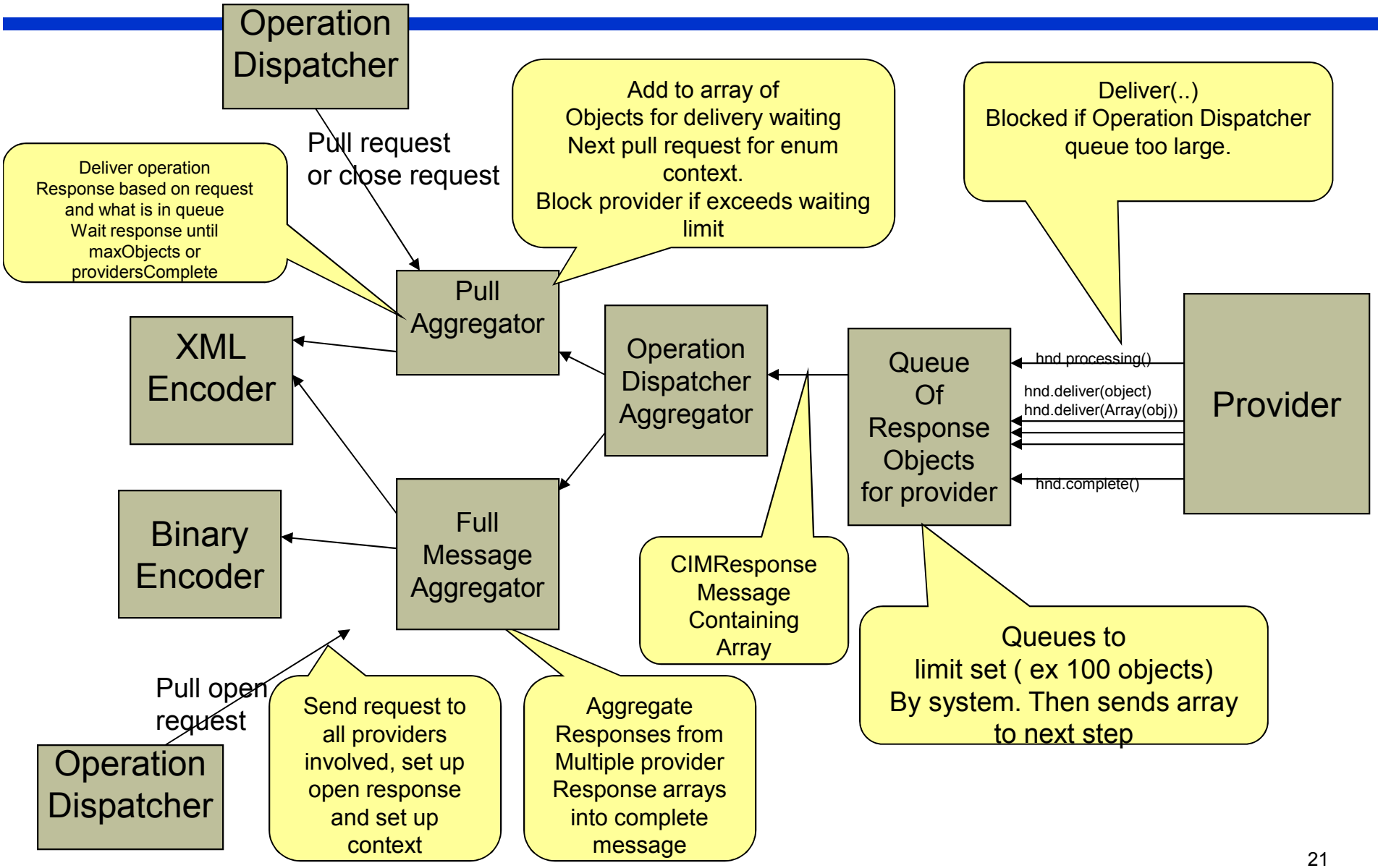
Enumerate Instances Example

```
void AssociationTestProvider::enumerateInstances(
    const OperationContext& context,
    const CIMObjectPath& classReference,
    const Boolean includeQualifiers,
    const Boolean includeClassOrigin,
    const CIMPropertyList& propertyList,
    InstanceResponseHandler& handler)
{
// Find the class corresponding to this instance
    CIMName reqClassName = classReference.getClassName();
    handler.processing();
    for (Uint32 i = 0; i < _classTable.size() ; i++)
    {
        if (reqClassName == _classTable[i].className)
        {
            CDEBUG("Class found " << reqClassName);
            for (Uint32 j = 0; j < _classTable[i].instances.size(); j++)
            {
                handler.deliver(_classTable[i].instances[j]);
            }
            handler.complete();
            return;
        }
    }
    throw CIMException(CIM_ERR_NOT_FOUND);
}
```

Original Response Flow through Pegasus



Pull Response Flow through Pegasus



Pull Operation provider interface changes

- Current Pegasus version
  - No changes to provider interface
- Future Pegasus update
  - Extend provider interface for request filter parameters.
  - Add mechanism to cleanly close provider delivery if pull operation closed.
  - Tied to CMPI Specification update.

MANAGEMENT DEVELOPERS CONFERENCE

Issues with delivery

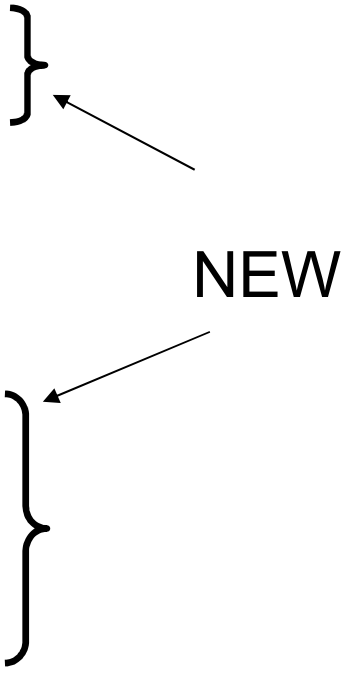
- Encourage providers to deliver small quantities with each call. This is good coding but not enforced.
  - Delivering large quantities in single call destroys Pegasus memory management model.
- Possible issues
  - Long blocking times on provider if client very slow. Pull designed to control flow, not let client play between pull calls.

Client API Similarities

- Follow same pattern of parameters but with new parameters attached.
- All possibly optional parameters are at end.
- All errors handled by exception as with existing model
- DIFFERENCES
  – Enumerates deliver namedInstances (with path)
  – OpenAssociators/OpenReferences return instances not objects. Providers will still create objects (sadly)

The Pegasus Pull Client API - 1

```
Array<CIMInstance> openEnumerateInstances(
    CIMEnumerationContext& enumerationContext,
    Boolean& endOfSequence,
    const CIMNamespaceName& nameSpace,
    const CIMName& className,
    const Boolean deepInheritance,
    const Boolean includeClassOrigin,
    const CIMPropertyList& propertyList = CIMPropertyList(),
    const String& filterQueryLanguage = String::EMPTY,
    const String& filterQuery = String::EMPTY,
    const Uint32Arg& operationTimeout  = Uint32Arg(),
    const Boolean continueOnError = false,
    const const Uint32Arg& maxObjectCount = Uint32Arg(0)
    );
```



Other Open… APIs parallel the Enumerate

Pegasus Pull Apis

```
// Pull Instances with Paths

Array<CIMInstance> pullInstancesWithPath
(
        CIMEnumerationContext& enumerationContext,
        Boolean& endOfSequence,
        const Uint32Arg& maxObjectCount = Uint32Arg(0)
);

 // Pull Instance Paths

 Array<CIMObjectPath> pullInstancePaths
(
        CIMEnumerationContext& enumerationContext,
        Boolean& endOfSequence,
        const Uint32Arg& maxObjectCount = Uint32Arg(0)
);
```

Close API

```
void closeEnumeration
(
        CIMEnumerationContext& enumerationContext
);
```

Simple Client Example

```
try
    {
        CIMNamespaceName nameSpace =
        "root/SampleProvider";
        String ClassName = "Sample_InstanceProviderClass";
        Boolean deepInheritance = false;
        Boolean includeClassOrigin = false;
        Uint32Arg maxObjectCount = 100;
        Boolean endOfSequence = false;
        Uint32Arg operationTimeout(0);
        Boolean continueOnError = false;
        String filterQueryLanguage = String::EMPTY;
        String filterQuery = String::EMPTY;
        Array<CIMInstance> cimInstances;
        CIMEnumerationContext ec;

        cimInstances = client.openEnumerateInstances(
            enumerationContext,
            endOfSequence,
            nameSpace,
            ClassName,
            deepInheritance,
            includeClassOrigin,
            CIMPropertyList(),
            filterQueryLanguage,
            filterQuery,
            operationTimeout,
            continueOnError,
            maxObjectCount );

        while (! endOfSequence)
        {
            Array<CIMInstance> cimInstancesTemp =
                client.pullInstancesWithPath(
                        enumerationContext,
                        endOfSequence,
                        maxObjectCount);


        cimInstances.appendArray(cimInstancesTe
        mp);
         }
    }
    catch (CIMException& e)
    {
        cerr << "CIMException Error: in
        testEnumerationWithPull "
            << e.getMessage() << endl;
        PEGASUS_TEST_ASSERT(false);
    }
    catch (Exception& e)
    {
        cerr << "Exception Error: in
        testEnumerationWithPull "
            << e.getMessage() << endl;
        PEGASUS_TEST_ASSERT(false);
    }
```

Limitations for 2.11

- Block concurrent close
  - Spec allows concurrency. We will not initially
- Server does not handle count operation
- May not include invokeMethod
  - Internal Pegasus support does not include invoke method
- No support for continue on error
  - Concerns about the effects, not the implementation
- May not include iterating client interface
- No support for Filters

Future Directions

- Pegasus
  - Extend so internal (cimom handle operations) use Pull
  - Possibly add count operation
  - Add InvokeMethod
  - Add Filters (After DMTF specification)
- DMTF Specification and Pegasus
  - Deprecate non-pull Instance operations
    - When DMTF deprecates them in specifications
  - Add specification for filters to the pull enumerations

MANAGEMENT DEVELOPERS CONFERENCE

Issues and Questions

- Server has capability to set several perfomance parameters ex.:
  - maxInteropTimeout
  - systemMaxObjectCount
  - maxConsecutiveZeroLengthPulls
- Proposal
  - These will be compile time options
- Question
  - Should any of these be runtime configuration
    - i.e could an adminstrator make any use of these?

MANAGEMENT DEVELOPERS CONFERENCE

Part 2.2
Provider Registration

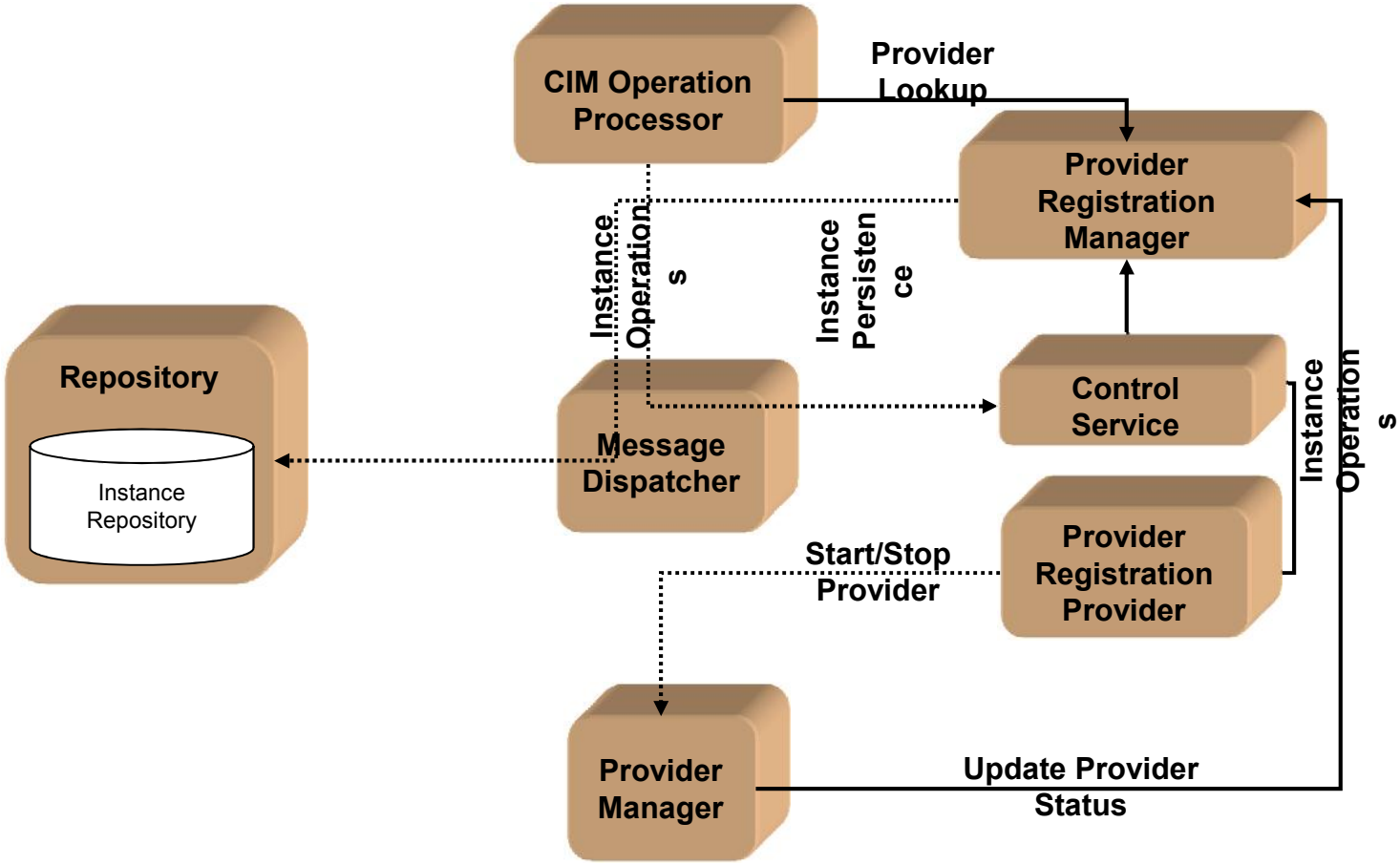
Provider Registration

- Not standardized in CIM today
  - Original concept was the “provider” qualifier
    - Used by some other CIM Servers today
  - Goal
    - Standard provider registration based on a provider registration profile
- Pegasus uses a set of classes to register providers
  - Create instances of provider registration classes (`PG_providermodule, PG_provider, PG_provider capabilities`)
  - Registration can be static or dynamic
    - Cimmof or cimmofl

Pegasus Provider Management

- **Provider Installation**
  - Put provider library into Pegasus provider directory
  - Register provider
- **Provider Registration**
  - Create instances of provider registration classes
  - Register by passing instances to Pegasus
    - Dynamic (cimmof)
    - Static (cimmofl)
- **Dynamic provider state control**
  - Enable / disable (`cimprovider` utility)

Provider Registration Service



Provider Registration Classes

See Schemas/Pegasus/Interop/VER20 PG_ProviderModule20.mof

- PG_Provider
  - Defines Provider module name (shared library)
  - Defines user context for module
  - Associates with Module & capabilities
- PG_ProviderModule
  - Defines module name for provider
  - Names provider
- PG_ProviderCapabilities
  - Points to provider and provider module
  - Defines provider type, Classname, etc.

Example, Instance Provider

```
instance of PG_ProviderModule
{
    Description = "Implements Sample_InstanceProviderClass";
    Caption = "Sample Pegasus Instance Provider Module";
    Name = "SampleInstanceProviderModule";
    Vendor = "OpenPegasus";
    Version = "2.0.0";
    InterfaceType = "C++Default";
    InterfaceVersion = "2.1.0";
    Location = "SampleInstanceProvider";
};

instance of PG_Provider
{
    ProviderModuleName = "SampleInstanceProviderModule";
    Name = "SampleInstanceProvider";
};

instance of PG_ProviderCapabilities
{
    ProviderModuleName = "SampleInstanceProviderModule";
    ProviderName = "SampleInstanceProvider";
    CapabilityID = "SampleInstanceProvider";
    ClassName = "Sample_InstanceProviderClass";
    Namespaces = { "root/SampleProvider" };
    ProviderType = { 2 };
    SupportedProperties = NULL;
    SupportedMethods = NULL;
};
```

Foreign keys link module, provider, capabilities.

Provider classname. One per capability

Limit to namespaces on this list.

Define types:
2 = Instance
3 = Assocation
4 = Indication
5 = method
6 = consumer
7 = instanceQuery

Limit properties And methods supported

Example Registration with Makefile

1. Create namespace and install any base classes required.
2. Compile the Schema for the provider to be registered
3. Register the provider by compiling the registration mof
4. NOTE: Normally the registration MOF is same name as Schema with “R”
   - Sample.mof SampleR.mof

```
cimmofl "-R$(REPOSITORY_DIR)" "-N$(REPOSITORY_NAME)"  \
        "-M$(REPOSITORY_MODE)“ \
        "-n$(INTEROPNS)" SampleProviderSchemaR.mof

OR
        cimmof "-n$(INTEROPNS)" SampleProviderSchemaR.mof
```

- See Examples in the sample and TestProvider Load directories
- Confirm registration with cimprovider utility.

MANAGEMENT DEVELOPERS CONFERENCE

Part 2.3
Handling CIM_Error And Standard Messages

Pegasus Status today

- CIM Error is supported
  - Pegasus Client
  - Pegasus Server (passing CIM_Errors as part of responses
  - Generating CIM_Error objects
  - Provider Interfaces – Generating CIM Errors
- Usage minimal
- Testing
  - End-end probably minimal because CIM_Error Class was experimental for a long time
  - Internal components part of standard test suite
- No work to date on standard messages

Pegasus Interface Extension

- Extended CIMException
  - Allows an array of CIM_Error objects to be attached to an exception
  - Server/Provider add CIM_Errors to an exception
  - Client takes them out if operation CIMException executed.
  - Client Driven Support
    - getErrorCount()
      - Return count of CIM_Error objects attached to exception
    - getError(index)
      - » Get the error at the defined index
  - Server and Provider driven support
    - AddError()
      - Adds a CIM_Error Object to an exception

C++ Provider example of CIM_Error

- Build instance of CIM_Error using pegasus CIMError class
- Define exception to include the CIMError or Array of CIMErrors
- Throw the exception
- NOTE: This uses the Server exception class, not the Provider exception class.

```
Const char * OwningEntityName = "OpenPegasus"
// Define instance of CIMError
CIMError ErrInst(OwningEntityName,
    "messageID",
    "This is the message",
    CIMError::PERCEIVED_SEVERITY_LOW,
    CIMError::PROBABLE_CAUSE_OTHER,
    CIMError::CIM_STATS_CODE_CIM_ERROR_FAILED);

// Create CIMException object
CIMException e(CIM_ERR_FAILED, "Test1 failed",
e.getInstance());

// Throw the exception
throw e;
```

Client Example

- **Getting CIM_Error objects from a response**

```
Try
{
    … Execute CIM Operation
}
Catch (CIMException& e)
{
      for (Uint32 I = 0 ; I < e.getErrorCount() ; i++
      CIMError err = e.getError(i);
     // . . . Process err
}
```

Conclusions

- CMPI extended for CIM_Error today
- C++ Providers can use CIMException extensions
- We can process multiple CIM_Errors through system (provider, server, client)
- No support internally std msg based specific errors

Core Objects

- Added new object as first class representation of CIM_Error
  - Src/Pegasus/General/CIMError.h /.cpp
  - Creates CIM_Error object
  - Provides getters and setters for all defined properties
  - Convert between CIMError C++ object and CIM_Error instance

MANAGEMENT DEVELOPERS CONFERENCE

Part 2.5
Debugging your Providers and Clients in The Pegasus Environment

Testing And Pegasus

- **Pegasus is well tested before release**
  - Unit tests, system tests, multiple system tests, cho (long run duration tests).
  - Head of all releasable CVS branches gets tested every night (ex. 2.8-branch, …, head)
- **Don't immediately assume it is a problem in Pegasus itself.**
- **Retest Pegasus itself through the pegasus/Makefile driven tests**
  - Make world or Make; make tests, etc.

Tools for Debugging

- **wbemexec** (line test)
  - *wbmexec –d2 test.xml*
  - (examples of .xml files are in tests/wetest)
- **cimcli**
  - Execute cim operations
- **Pegasus Logs**
- **Pegasus Traces**
- **Pegasus Client trace**
- Debuggers
  - Gdb
  - Visual Studio
- Memory Tools
  - valgrind

MANAGEMENT DEVELOPERS CONFERENCE

Wbemexec

- **Issue requests to pegasus as xml**
- **Display xml responses**
- **Ex**
  - *wbmexec –d2 test.xml*
- ***Many examples in source code***
  - (see tests/wetest)
- Pro
  - Test at xml level. Use to define xml issues
- Con
  - Low level, difficult to create tests

XML request sample

```
<?xml version="1.0" ?>
<CIM CIMVERSION="2.0" DTDVERSION="2.0">
 <MESSAGE ID="50000" PROTOCOLVERSION="1.0">
  <SIMPLEREQ>
   <IMETHODCALL NAME="EnumerateInstanceNames">
    <LOCALNAMESPACEPATH>
     <NAMESPACE NAME=“root"/>
     <NAMESPACE NAME=“cimv2"/>
    </LOCALNAMESPACEPATH>
    <IPARAMVALUE NAME="ClassName">
     <CLASSNAME NAME=“PG_OperatingSystem"/>
    </IPARAMVALUE>
   </IMETHODCALL>
  </SIMPLEREQ>
 </MESSAGE>
</CIM>
```

cimcli – execute operations

- Interactive CLI client that executes CIM Operations
- Through 2.9
  - Implements all of the read operations and simplistic invoke method
- 2.10
  - Implements Create, Modify, instance, correct invokeMethod, testing responses, etc.
- Examples
  - `cimcli ni Person –enumerateinstance names`
  - `cimcli ci Person ssn=1 first=karl last=schopmeyer`

Cimcli (cont)

- Includes
  - All CIM-XML operations except create/modify class
  - Some more general actions
    - Get namespaces
    - Get all instanceNames in a namespace
    - Test instance against command line definition
- Limitations
  - Command line

CIM Listeners

- Example code exists not a complete display listener today
- Note: SimpleWbem has a command line cimlistener that can be used.

Other test clients

- CIMNavigator
- CIMSurfer
- WSI client browser
- SNIA client browser
- pywbem

MANAGEMENT DEVELOPERS CONFERENCE

Logs

- Pegasus generates production log output
- Set logs to max level to get the most info
- Log destination is system dependent
  - Windows – log files
  - *nux – syslog
- Primarily production level issues
- BUT: In many cases the logs will tell you what the problem is. Look at them

Pegasus Trace

- This is the core debugging tool in Pegasus
- Pegasus CIM Server is throughly instrumented for trace output.
- Tracing is command line controllable
- There is now a memory based circular trace
  - Lowers impact on server
  - Avoids the enormous files that can occur with disk trace.
- See PEP 315 & 316 For details

Setting Up for Trace

- Run Pegasus in the forground, not as a service or daemon
  - Windows – **`cimserver -help`**
  - Linux – **`cimserver daemon=false`**
- Run with providers in-process
  - Easier to debug than separate processes
  - Typical good settings
    - **`export PEGASUS_DEFAULT_ENABLE_OOP=false`**
    - **`export PEGASUS_DISABLE_PRIVILEGED_TESTS=true`**
    - **`export PEGASUS_DISABLE_PROV_USERCTXT=true`**
- Set the trace level and components
  - Either permanent or on startup
- Isolate the action that is a problem and execute this action by itself with trace

How to Generate Trace

- Set the trace component:
  - **`bin/cimconfig -s traceComponents=Thread,ProvManager`**
- Logs the data in cimserver.trc (default) file
  - Or file defined by config variable `traceFilePath`
- Set the trace level:
  - **`bin/cimconfig -s traceLevel=4`**
- Or set trace for current server start
  - **`Cimserver traceComponents=All traceLevel=4`**
- See also mak/Buildmakefile for typical trace configurations.

Trace Levels

- Each trace call has an associated level
- Different levels per trace (pre Pegasus 2.8)
  - 0 – Tracing off (default)
  - 1 - Function Entry/Exit
  - 2 - Basic flow trace messages, low data detail
  - 3 - Inter-function logic flow, medium data detail
  - 4 - High data detail
- Levels Post 2.8 – Separated Entry/exit
  - 0 – Tracing off (default)
  - 1 – Severe and log messages
  - 2 - Basic flow trace messages, low data detail
  - 3 - Inter-function logic flow, medium data detail
  - 4 - High data detail
  - 5 – High data detail + Function Entry/Exit

List of Trace Components (2.10)

- racing is done per server component (not per source file).
  - Xml
  - **XmlIO**
  - Http
  - Repository
  - Dispatcher
  - OsAbstraction
  - Config
  - IndicationHandler
  - Authentication
  - Authorization
  - UserManager
  - Shutdown
  - Server
  - IndicationService
  - MessageQueueService
  - ProviderManager
  - ObjectResolution
  - WQL
  - CQL
  - Thread
  - CIMExportRequestDispatcher
  - SSL
  - CIMOMHandle
  - L10N
  - ExportClient
  - Listener
  - **DiscardedData**
  - ProviderAgent
  - IndicationFormatter
  - StatisticalData
  - CMPIProvider
  - IndicationGeneration
  - IndicationReceipt
  - CMPIProviderInterface
  - WsmServer
  - **LogMessages**
  - WMIMapperConsumer
  - ControlProvider

See src/Pegasus/Common/Tracer.cpp

Memory based trace

- Started 2.8 or 2.9 (See PEP 316)
  - Circular cache in core
  - Configuration variables
    - **`traceMemoryBufferKbytes=<`*size of in-memory buffer in kB*`>`**
    - **`traceFacility= (file,memory ,`** log)
- If this memory is part of a dump the trace messages can be found by the eye-catcher "PEGASUSMEMTRACE" at the top of the memory buffer. The trace is in clear text and the last written message has the suffix "EOTRACE".
- I think it also dumps the buffer on pegasus exit

MANAGEMENT DEVELOPERS CONFERENCE

Notes on reviewing trace

- Always trace the io (XmlIO) and discardedData
  - XmlIO frames the rest of the trace
  - You can see what is coming and going
  - discardedData tells you when we throw things away
- Don't trace function calls at first.
  - Look at the data, not the flow
- If there are problems, look at the trace in the area where the problem is occurring
  - Look for keywords that could represent the issue
    - Exception, error, etc.

Limitations

- We don’t support selective provider tracing.
- You can add traces to your provider but it all goes into one big category
- It helps to understand the overall architecture since this is the basis for the component definition.

How to Understand Traces

- The major goals of tracing are:
  - Confirm what is actually entering and leaving the server
  - See what providers are actually called
  - Determine the data (operation, etc)flow through the CIM Server
  - Try to isolate what component made the decision that impacts your issue

MANAGEMENT DEVELOPERS CONFERENCE

Trace Limitations

- High volume.
  - Multi gb trace files are common
- Traces all functions
  - The function trace has only a single level
- Developer oriented
  - Without the source following much of the trace is difficult (Except XmlIO)
  - BUT – XmlIO, dispatcher, providerManager define operation flow and most data

And after the trace, What??

- Here is where the fun begins
  - Debuggers
  - Core dumps
  - Adding Trace points yourself
  - Finally the dreaded printf(…)
  - Specialized debug support
    - Special malloc testers
    - GNU exception backtrace
    - ….

Pegasus Client trace

- Conditional compile in CIMClient.cpp
  - `export PEGASUS_CLIENT_TRACE_ENABLE=true`
  - Compile pegasus/src/Pegasus/Client
- When you run a client (ex. cimcli)
  - `source export PEGASUS_CLIENT_TRACE=both:both`
  - Then execute your client request:ex. `cimcli ni myclass`
- Will generate requests and responses directly to console.

MANAGEMENT DEVELOPERS CONFERENCE

Memory Issues

- Commercial and OpenSource Tools
  - Valgrind can be your friend
  - We use valgrind extensively (memcheck)
- Regular tests of Pegasus against valgrind
  - Nightly Pegasus tests
- First confirm Pegasus with std operations
  - Then test your operation

Provider Only valgrind

- Build with out-of-process providers
- Replace cimprovagt with valgrind script

```
#!/bin/sh
## Original Author: Tim Potter
# move cimprovagt to cimprovagt.real
# move this file to cimprovagt
#
# mv /usr/sbin/cimprovagt /usr/sbin/cimprovagt.real
# cp cimprovagt.wrapper /usr/sbin/cimprovagt

## By default the script doesn't call valgrind - enable it by
## creating a semaphore file of the form /tmp/$MODULE.valgrind where module
## is the module name in the output of "cimprovider -l".

## Or create a file /tmp/LogAll.valgrind which valgrinds all providers.

module=$5

VALGRIND_ARGS="--leak-check=yes --trace-children=yes --log-file=/tmp/$module.valgrind"

if [ -e /tmp/$module.valgrind -o -e /tmp/LogAll.valgrind ]; then
        exec /usr/bin/valgrind $VALGRIND_ARGS \
                        /usr/sbin/cimprovagt.real "$@"
else
        exec /usr/sbin/cimprovagt.real "$@"
fi
```

Finding Server Crashes

- Yes we sometimes get server crashes
- Dumps
  - Turn on dump if possible
  - Set debugging mode if possible
  - Use the dumps and get stack trace info
  - Communicate your issue with other pegasus users
- Memory based trace
  - This can catch the last few server actions
  - There is an extra load cost but not major
- Try to isolate your problem to a single operation

Questions & Discussion

We would like to get your feedback on issues, priorities, users/usage, requests for OpenPegasus.