R"""Capsule and capsule proxy class

Author : Anders Andersen
Created On : Thu Oct 29 20:29:57 1998
Last Modified By: Anders Andersen
Last Modified On: Fri Sep 10 13:32:05 1999
Status : Unknown, Use with caution!

The implementation of the capsule and the capsule proxy class. See `capsule.py` for an overview and the classes below for more detailed information. Below is an example of setting up a server (implemented with the `Server` class) providing the interface `"req"` (see information about the nameserver module in the `nameserver.py` file).

```
from socket import gethostname
from lbind import *
from component import *
from nameserver import *
import capsule

class Server:
    ...

server = capsule.local.mkComponent(componentFactory,(["req"],Server))
name = "req %s" % (gethostname(),)
ns = NameServerProxy(host,port)
ns.exportIRef(name,capsule.local.getIRef(server,"req"))

capsule.local.serve()
```

"""

```
# Mainly to grab information about exceptions
import sys

# For low level communication
from socket import *

# We need to check the type of some attributes
from types import *

# Misc values for the Open-ORB core
from misc import *

# Local bindings (and interface references)
from lbind import *


class CapsuleException(OpenORBException):
```

R"""Capsule exception

All exceptions or errors introduced by the `capsule` module is handled by this exception class.

"""

```
    pass


class Capsule:
```

```
R"""The capsule
```

The capsule is the support environment for objects and components in our model. It provide support for local and remote components (remote only if the serve thread is running). It can be used to create and register components, delete components and create local bindings between interfaces of registered components in the capsule. It also provides a low level feature for calling methods of registered components in the capsule. You can use an instance of the `CapsuleProxy` class to access the capsule remotely (again, if the serve thread is running).

```
"""

def __init__(self):
    R"""Initialize the capsule
```

Set the initial state of capsule.

```
    """
    from msg import Msg, PackException
    import nodemngr
    self.nm = nodemngr.nm
    self.message = Msg("", self.nm.newPort("capsule %s" % ('self',)))
    self.listen = None
    self.components = {}

def __del__(self):
    R"""Delete the capsule
```

Release resources if the capsule is deleted. This is also used when the serve thread is terminated.

```
    """
    debug("Capsule at %s (%d) deleted" % (gethostname(), self.message.port))
    if self.message:
        if self.message.port:
            try:
                self.nm.delPort(self.message.port)
            except:
                pass
        del self.message
    if self.listen:
        del self.listen

def __fetchIRef__(self, riref):
    R"""Fetch an interface reference
```

Fetch the actual interface reference from a remote interface reference. The interface must be a part of a registered component in this capsule.

```
    """
    object = riref.__local__["object"]
    if type(object) is DictType:
        return self.__fetchIRef__(
            self.components[object["comp"]].interfaces[object["iface"]])
    else:
        return riref

def __serveloop__(self):
    R"""The capsule server loop
```

The capsule server loop is either started with the `serve` or the `servethread` method (the `servethread` method runs the sever loop in a separate thread). The server loop responds to remote request to this capsule.

```
    """

    # The main (serving) loop
```

```
        while 1:

            # Recieve a request
            connection, requests = self.listen.recvreq()
            for req in requests:
                debug("Capsule request: %s" % (`req`,))
                if req["op"] == "stopserve":
                    del self.listen
                    return

                # Perform the request and send a reply (possible an error)
                if not req.has_key("args"): req["args"] = ()
                if not req.has_key("kw"): req["kw"] = {}
                if not req.has_key("announce"): req["announce"] = 0
                if not req.has_key("thread"): req["thread"] = 0
                if req["announce"]:
                    connection.close()
                    try:
                        if req["thread"]:
                            import thread
                            thread.start_new_thread(
                                self.op[req["op"]], req["args"], req["kw"])
                        else:
                            apply(self.op[req["op"]], req["args"], req["kw"])
                    except Exception:
                        (exc, val, tb) = sys.exc_info()
                        debug("Capsule serve error (ignored)")
                        debug_exc(exc, val, tb)
                else:
                    try:
                        rep = apply(self.op[req["op"]], req["args"], req["kw"])
                    except Exception:
                        (exc, val, tb) = sys.exc_info()
                        debug("Capsule serve error")
                        self.listen.sendrep(connection, ErrorObject(exc, val, tb))
                    else:
                        self.listen.sendrep(connection, rep)

    def __serve__(self, threaded=1):
        R"""Start serving

        Prepare for the serve loop, and then start it.

        """

        # Initialize data structures
        debug("Capsule %s (%d) ready to serve" %
              (gethostname(), self.message.port))
        self.op = {"registerComponent": self.registerComponent,
                   "mkComponent": self.mkComponent,
                   "rcpComponent": self.rcpComponent,
                   "delComponent": self.delComponent,
                   "callMethod": self.callMethod,
                   "getIRef": self.getIRef,
                   "localBind": self.localBind,
                   "localBindOneWay": self.localBindOneWay,
                   "breakBinding": self.breakBinding,
                   "newPort": self.newPort,
                   "delPort": self.delPort}
        # Create and initialize listen object
        from msg import Msg
        self.listen = Msg(self.message.node, self.message.port, 1)

        # Start serve loop (threaded or not)
        if threaded:
            import thread
            thread.start_new_thread(self.__serveloop__, (), {})
        else:
            self.__serveloop__()

    def serve(self):
        R"""Start the server loop

        Start the server loop without creating a new thread. This is not normally recommended if threads
        are supported on your platform.

        """
        self.__serve__(0)

    def servethread(self):
        R"""Start the server loop in a new thread

        Start the server loop by creating a new thread. The new thread terminates silently when the server
        loop is terminated.

        """
        self.__serve__(1)

    def stopserve(self):
        R"""Stop the serve loop

        Sends a message to the serve loop asking it to terminate.

        """
        self.message.announce({"op": "stopserve", "announce": 1})

    def registerComponent(self, comp):
        R"""Register a component

        Register a component in this capsule using the representation string as the key. The key is returned.

        """
        self.components[`comp`] = comp
        debug("Capsule registerComponent: %s" % (`comp`,))
        return `comp`

    def mkComponent(self, mkcomp=None, args=(), kw={}):
        R"""Create a component

        Create a component in this capsule using the mkcomp function (either a component factory or a
        component class).

        """
        debug("Capsule mkComponent: %s:%s,%s" % (`mkcomp`, `args`, `kw`))
        try:
            comp = apply(mkcomp, args, kw)
        except Exception:
            (exc, val, tb) = sys.exc_info()
            str = "Capsule mkComponent: mkcomp failed"
            debug(str)
            debug_exc(exc, val, tb)
            raise CapsuleException, str
        return self.registerComponent(comp)
    def rcpComponent(self, comp, capsule):
        R"""Remote copy of component

        Copy the component to the given (remote) capsule.

        """
        remoteCapsule = CapsuleProxy(capsule.node, capsule.port)
        try:
            return remoteCapsule.registerComponent(self.components[comp])
        except KeyError, val:
            str = "Capsule rcpComponent: %s doesn't exists (%s)" % (comp, 'val')
            raise KeyError, str

    def delComponent(self, comp):
        R"""Delete component

        Delete a registered component from the capsule.

        """
        try:
            del self.components[comp]
        except KeyError, val:
            str = "Capsule delComponent: %s doesn't exists: %s" % (comp, 'val')
            raise KeyError, str

    def callMethod(self, comp, iface, method, args, kw):
        R"""Call a method

        Call a method of an interface of a registered component.

        """
        debug("Capsule callMethod: %s->%s->%s" % (comp, iface, method))
        try:
            return apply(getattr(
                    self.components[comp].interfaces[iface].__local__["iobj"],
                    method), args, kw)
        except Exception:
            (exc, val, tb) = sys.exc_info()
            str = "Capsule callMethod: unable to call %s (%s->%s)" % (
                method, comp, iface)
            debug(str)
            debug_exc(exc, val, tb)
            raise CapsuleException, str

    def announceMethod(self, comp, iface, method, args, kw):
        R"""Call a method without a reply

        Call a method in the capsule but don't expect a reply.

        """

        self.callMethod(comp, iface, method, args, kw)

    def announceThread(self, comp, iface, method, args, kw):
        R"""Start a thread

        Start a new thread which runs the given method in the capsule. Any results will not be returned.

        """
        thread.start_new_thread(self.callMethod,
                                (comp, iface, method, args, kw))

    def sendMethod(self, comp, iface, method, args, kw):
        R"""Call a method but collect the result later

        Call a method in the capsule, but the result can be collected later with recvMethod.

        """
        return self.callMethod(comp, iface, method, args, kw)

    def recvMethod(self, message):
        R"""Collect the result of an earlier method call

        Collect the result of an earlier method call to the capsule.

        """
        return message

    def getIRef(self, comp, iface):
        R"""Get an interface reference

        Get an interface reference from a registered component.

        """
        try:
            iref = self.components[comp].interfaces[iface]
        except KeyError, val:
            str = "Capsule getIRef: iref (%s,%s) not found (%s)" \
                  % (comp, iface, ‘val‘,)
            raise CapsuleException, str
        if type(iref.__local__["object"]) is DictType:
            obj = iref.__local__["object"]
        else:
            obj = {"capsule": self , "comp": comp, "iface": iface}
        return IRef(obj, iref.__expID__, iref.__impID__)

    def localBind(self, riref1, riref2):
        R"""Create a local binding

        Create a local binding between interfaces of two registered components in this capsule.

        """
        try:
            iref1 = self.__fetchIRef__(riref1)
            iref2 = self.__fetchIRef__(riref2)
        except AttributeError, val:
            str = "localBind: AttributeError: %s" % (‘val‘,)
            raise AttributeError, str
        except KeyError, val:
            str = "localBind: KeyError: %s" % (‘val‘,)
            raise KeyError, str
        iref1.__testImpInterface__(iref2)
        iref2.__testImpInterface__(iref1)
        return LBindCtrl(None, iref1, iref2)

    def localBindOneWay(self, riref1, riref2):
        R"""Create a one-way local binding

        Create a one-way local binding between interfaces of two registered components in this capsule.

        """
        try:
            iref1 = self.__fetchIRef__(riref1)
            iref2 = self.__fetchIRef__(riref2)
        except AttributeError, val:
            str = "localBindOneWay: AttributeError: %s" % (‘val‘,)
            raise AttributeError, str
        except KeyError, val:
            str = "localBindOneWay: KeyError: %s" % ('val',)
            raise KeyError, str
        iref2.__testImpInterface__(iref1)
        return LBindCtrl(None, iref1, iref2)

    def breakBinding(self, riref1, riref2):
        R"""Break a binding

        Break a binding between the interfaces of two registered components in this capsule.

        """
        try:
            iref1 = self.__fetchIRef__(riref1)
            iref2 = self.__fetchIRef__(riref2)
        except AttributeError, val:
            str = "breakBinding: AttributeError: %s" % ('val',)
            raise AttributeError, str
        except KeyError, val:
            str = "breakBinding: KeyError: %s" % ('val',)
            raise KeyError, str
        if ((iref1.__local__["object"] == iref2.__remote__["object"]) or
            (iref2.__local__["object"] == iref1.__remote__["object"])):
            iref1.__breakBinding__()
            iref2.__breakBinding__()
        else:
            raise CapsuleException, \
                  "breakBinding: can not break binding between local and " + \
                  "non-local irefs"

    def newPort(self, info):
        R"""Get a new communication port

        Get a new communication port. Communication port management is done by the node manager,
        and the request is forwarded to it (the users shouldn't be aware of the node manager).

        """
        return self.nm.newPort(info)

    def delPort(self, port):
        R"""Release a communication port

        Delete (or release) a communication port. Communication port management is done by the node
        manager, and the request is forwarded to it.

        """
        return self.nm.delPort(port)


class CapsuleProxy:
    R"""The capsule proxy

    Instances of the capsule proxy are created in remote (other) capsules to access services of a capsule (running
    the serve loop).

    """

    def __init__(self, node="", port=0):
        R"""Initialize the capsule proxy

        Save information about the remote capsule.

        """
        from msg import Msg
        self.message = Msg(node, port)

    def stopserve(self):
        R"""Stop server loop

        Terminate the server loop in the remote capsule. The result is that you can not access the remote
        capsule anymore (and in may cases will it also terminates the capsule).

        """
        self.message.announce({"op": "stopserve", "announce": 1})

    def registerComponent(self, comp):
        R"""Register a component

        Register a component in the capsule. The key is returned.

        """
        return self.message.message(
            {"op": "registerComponent", "args": (comp,)})

    def mkComponent(self, mkcomp=None, args=(), kw={}):
        R"""Create a component

        Create a component in the remote capsule using the given component factory (or class).

        """
        return self.message.message(
            {"op": "mkComponent", "args": (mkcomp, args, kw)})

    def rcpComponent(self, comp, capsule):
        R"""Remote copy of component

        Copy the component to a (remote) capsule.

        """
        return self.message.message(
            {"op": "rcpComponent", "args": (comp, capsule)})

    def delComponent(self, comp):
        R"""Delete a component

        Delete the given component in the remote capsule.

        """
        self.message.message({"op": "delComponent", "args": (comp,)})

    def announceMethod(self, comp="", iface="", method="", args=(), kw={}):
        R"""Call a method without a reply

        Call a method in the remote capsule but don't expect a reply.

        """
        self.message.announce({"op": "callMethod", "announce": 1,
                               "args": (comp, iface, method, args, kw)})

    def announceThread(self, comp="", iface="", method="", args=(), kw={}):
        R"""Start a thread

        Start a new thread which runs the given method in the remote capsule. Any results will not be
        returned.

        """
        self.message.announce({"op": "callMethod", "announce": 1, "thread": 1,
                               "args": (comp, iface, method, args, kw)})

    def sendMethod(self, comp="", iface="", method="", args=(), kw={}):
        R"""Call a method but collect the result later

        Call a method in the remote capsule, but the result can be collected later with recvMethod.

        """
        return self.message.sendreq({"op": "callMethod",
                                     "args": (comp, iface, method, args, kw)})

    def recvMethod(self, message):
        R"""Collect the result of an earlier method call

        Collect the result of an earlier method call to the remote capsule.

        """
        return self.message.recvrep(message)

    def callMethod(self, comp="", iface="", method="", args=(), kw={}):
        R"""Call a method

        Call a method in the remote capsule. Wait for the result and return it.

        """
        return self.message.message(
            {"op": "callMethod", "args": (comp, iface, method, args, kw)})

    def getIRef(self, comp="", iface=""):
        R"""Get an interface reference

        Get an interface reference from a registered component in the remote capsule.

        """
        return self.message.message({"op": "getIRef", "args": (comp, iface)})

    def localBind(self, iref1, iref2):
        R"""Create a local binding

        Create a local binding in the remote capsule.

        """
        if equalCapsule(iref1, iref2):
            self.message.message({"op": "localBind", "args": (iref1, iref2)})
            return LBindCtrl(self, iref1, iref2)
        else:
            raise CapsuleException, \
                  "Local bind only between interfaces in the same capsule"

    def localBindOneWay(self, iref1, iref2):
        R"""Create a one-way local binding

        Create a one-way local binding in the remote capsule.

        """
        if equalCapsule(iref1, iref2):
            self.message.message({"op": "localBindOneWay", "args": (iref1, iref2)})
            return LBindCtrl(self, iref1, iref2)
        else:
            raise CapsuleException, \
                  "Local bind only between interfaces in the same capsule"

    def breakBinding(self, iref1, iref2):
        R"""Break a binding

        Break a binding in the remote capsule.

        """
        self.message.message({"op": "breakBinding", "args": (iref1, iref2)})

    def newPort(self, info):
        R"""Request a new communication port

        Request a new communication port (indirectly) from the node manager of the remote capsule.

        """
        return self.message.message({"op": "newPort", "args": (info,)})

    def delPort(self, port):
        R"""Release a communication port

        Release a communication port (indirectly) at the node manager of the remote capsule.

        """
        self.message.message({"op": "delPort", "args": (port,)})


def equalCapsule(iref1, iref2):
    R"""Interface references in the same capsule?

    Are these interface references in the same capsule (identified by node and communication port)?

    """

    # Get capsule (or capsule proxy)
    if type(iref1.__local__["object"]) is DictType:
        capsule1 = iref1.__local__["object"]["capsule"]
    else:
        capsule1 = local
    if type(iref2.__local__["object"]) is DictType:
        capsule2 = iref2.__local__["object"]["capsule"]
    else:
        capsule2 = local

    # Return true if capsule1 = capsule2
    return (capsule1.message.node == capsule2.message.node and
            capsule1.message.port == capsule2.message.port)
```