R"""Message handling

Author : Anders Andersen
Created On : Tue Oct 27 09:54:46 1998
Last Modified By: Anders Andersen
Last Modified On: Wed Feb 25 15:20:03 2004
Status : Unknown, Use with caution!



The pack and unpack functions are used to pack and unpack objects sent between different capsules. These functions use strip to prepare the objects to be packed and unstrip to restore the unpacked objects. The Msg class provides a set of methods used to implement different types of synchronuos and asynchronuos messsage passing methods (remote method calls) over TCP/IP. The FlowSrc and FlowSink classes implements a flow (stream) over an UDP/IP connection. No buffering, ordering or retransmission is done.

"""

```
# System modules
import re
import inspect
import cPickle


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


# Errors and exceptions

class PackException(OpenORBException):
    R"""A pack exception

    Alle exceptions or errors in the packing and unpacking process generates an PackException.

    """
    pass


# Get the node's IP number

# Regular expression for IP numbers
ipnumexpr = re.compile(r'\d{1,3}\.\d{1,3}\.\d{1,3}\.\d{1,3}')

def fetchnode(node=None):
    R"""Get the IP number

    Get the IP numer from the given host name node. The node argument can either be a host name, an IP numer or empty (not given). If the node argument is missing the IP numer of the local host is returned.

    """
    if not node:
        return gethostbyname(gethostname())
    elif ipnumexpr.match(node):
        return node
    else:
        return gethostbyname(node)
```

```
# Code for strip and unstrip

class ClassInfo:
    R"""Information about a class

    This class is used to create objects that contains information about a class.
    """
    def __init__(self, name="", code="", bases=(), module=""):
        self.name=name
        self.code=code
        self.bases=bases
        self.module=module

def strip(object):
    R"""Strip the object

    We have to prepare objects to be pickled.
    """

    # Strip each element in lists
    if type(object) is ListType:
        return map(strip, object)

    # Strip each element in tuples
    if type(object) is TupleType:
        return tuple(map(strip, list(object)))

    # Strip each element in dictionaries
    if type(object) is DictType:
        for (key, item) in object.items():
            object[key] = strip(item)
        return object

    # Classes are transfered as instances of ClassInfo
    if type(object) is ClassType:
        return ClassInfo(name=object.__name__,
                         code=inspect.getsource(inspect.getmodule(object)),
                         bases=strip(object.__bases__),
                         module=object.__module__)

    # Duplicate interface references
    if isinstance(object, IRef):
        if type(object.__local__["object"]) is DictType:
            obj = {}
            for (key, item) in object.__local__["object"].items():
                obj[key] = strip(item)
            return IRef(obj, object.__expID__, object.__impID__)
        else:
            return IRef(None, object.__expID__, object.__impID__)

    # Duplicate local bind control objects
    if isinstance(object, LBindCtrl):
        return LBindCtrl(
            strip(object.capsule), strip(object.iref1), strip(object.iref2))

    # Duplicate capsule proxies and make capsule proxies for capsules
    if isinstance(object, CapsuleProxy) or isinstance(object, Capsule):
        return CapsuleProxy(object.message.node, object.message.port)
```

```
    # Don't strip
    return object

def unstrip(object):
    R"""Unstrip the object

    We may have to restore some unpickled objects.

    """

    # Unstrip each element in lists
    if type(object) is ListType:
        return map(unstrip, object)

    # Unstrip each element in tuples
    if type(object) is TupleType:
        return tuple(map(unstrip, list(object)))

    # Unstrip each element in dictionaries
    if type(object) is DictType:
        for (key, item) in object.items():
            object[key] = unstrip(item)
        return object

    # Unstrip classes (strip have made all clasess to ClassInfo objects)
    if isinstance(object, ClassInfo):
        exec object.code in globals()
        print 'eval(object.name)'
        return eval(object.name)

    # Others are unchanged
    return object


def pack(object):
    R"""Pack arguments

    Pack the arguments with cPickle.

    """
    return cPickle.dumps(strip(object))

def unpack(str):
    R"""Unpack arguments

    Unpack the arguments with cPickle.

    """
    return unstrip(cPickle.loads(str))


# Use marshal to serialize (and deserialize) data in a flow
import marshal
serialize = marshal.dumps
deserialize = marshal.loads


# For low level communication (implementation of the message and flow classes)
from socket import *
import string

class RecvMsg:
```

```
    def __init__(self):
        self.buffers = {}
    def __call__(self, msgsocket, single=0):
        if self.buffers.has_key('msgsocket'):
            msg = self.buffers['msgsocket'] + msgsocket.recv(BUFSIZ)
            del self.buffers['msgsocket']
        else:
            msg = msgsocket.recv(BUFSIZ)

        while 1:
            try:
                (sz, pk) = string.split(msg, ":", 1)
            except ValueError:
                msg = msg + msgsocket.recv(BUFSIZ)
            else:
                break

        size = string.atoi(sz)
        rs = len(pk)
        while rs < size:
            pk = pk + msgsocket.recv(BUFSIZ)
            rs = len(pk)
        if rs > size:
            if single:
                self.buffers['msgsocket'] = pk[size:]
                return pk[:size]
            else:
                pklist = []
                while rs >= size:
                    pklist.append(pk[:size])
                    msg = pk[size:]
                    try:
                        (sz, pk) = string.split(msg, ":", 1)    #!
                    except ValueError:
                        size = 0
                        pk = msg
                        rs = len(pk)
                        break
                    else:
                        size = string.atoi(sz)
                        rs = len(pk)
                if rs > 0:
                    self.buffers['msgsocket'] = pk
                return pklist
        if single:
            return pk
        else:
            return [pk]

recvmsg = RecvMsg()

def sendmsg(msgsocket, pk):
    msgsocket.send("%d:" % (len(pk),) + pk)

class Msg:
    R"""Message handling

    Instances of the message class provides methods for sending and receiving messages.

    """
```

```
257
def __init__(self, node="", port=0, listen=0):                          258
    R"""Initialize instance of Msg class                                259

    Save the node and communication port information.

    """
    self.node = fetchnode(node)                                         264
    self.port = port                                                    265
    if listen:                                                          266
        self.listensocket = socket(AF_INET, SOCK_STREAM)                267
        self.listensocket.bind((self.node, self.port))                  268
        self.listensocket.listen(1)                                     269
    else:                                                               270
        self.listensocket = None                                        271
                                                                        272
def __del__(self):                                                      273
    R"""Clean up                                                        274

    Clean up when this object is deleted.

    """
    if self.listensocket:                                               279
        self.listensocket.close()                                       280
        self.listensocket = None                                        281
                                                                        282
def sendreq(self, req):                                                 283
    R"""Sends a request                                                 284

    Sends a request to node and port saved earlier.

    """
    debug("Msg sendreq (%s:%d): %s" % (self.node, self.port, 'req'))    289
    try:                                                                290
        msgsocket = socket(AF_INET, SOCK_STREAM)                        291
    except error, str:                                                  292
        debug("Msg sendreq unable to create socket: %s" % (str,))       293
        raise error, str                                                294
    try:                                                                295
        msgsocket.connect((self.node, self.port))                       296
    except error, str:                                                  297
        debug("Msg sendreq unable to connect: %s" % (str,))             298
        raise error, str                                                299
    try:                                                                300
        sendmsg(msgsocket, pack(req))                                   301
    except error, str:                                                  302
        debug("Msg sendreq unable to send: %s" % (str,))                303
        raise error, str                                                304
    except PackException, str:                                          305
        debug("Msg sendreq unable to pack: %s" % (str,))                306
        raise PackException, str                                        307
    return msgsocket                                                    308
                                                                        309
def recvreq(self):                                                      310
    """Receive a request                                                311

    Receive a request on the listen socket

    """
    debug("Msg recvreq waiting (%s:%d)" % (self.node, self.port))       316
    try:                                                                317
        msgsocket, addr = self.listensocket.accept()                    318
    except error, str:                                                  319
        debug("Msg recvreq unable to accept: %s" % (str,))              320
```

```
            raise error, str
        try:
            requests = []
            for req in recvmsg(msgsocket):
                requests.append(unpack(req))
                debug("Msg recvreq: %s" % (‘unpack(req)‘,))
        except error, str:
            debug("Msg recvreq unable to receive: %s" % (str,))
            raise error, str
        except PackException, str:
            debug("Msg recvreq unable to unpack: %s" % (str,))
            raise PackException, str
        return (msgsocket, requests)

    def sendrep(self, msgsocket, rep):
        """Send a reply

        Send a reply on an earlier received request.

        """
        debug("Msg sendrep (%s:%d): %s" % (self.node, self.port, ‘rep‘))
        try:
            sendmsg(msgsocket, pack(rep))
            msgsocket.close()
        except error, str:
            debug("Msg sendrep unable to send: %s" % (str,))
            raise error, str
        except PackException, str:
            debug("Msg sendrep unable to pack: %s" % (str,))
            raise PackException, str

    def recvrep(self, msgsocket):
        R"""Receive a reply

        Receive a reply from an earlier request.

        """
        debug("Msg recvrep waiting (%s:%d)" % (self.node, self.port))
        try:
            rep = unpack(recvmsg(msgsocket,1))
            debug("Msg recvrep: %s" % (‘rep‘,))
        except error, str:
            debug("Msg recvrep unable to receive: %s" % (str,))
            raise error, str
        except PackException, str:
            debug("Msg recvrep unable to unpack: %s" % (str,))
            raise PackException, str
        msgsocket.close()
        if isinstance(rep, ErrorObject):
            tb = ""
            for tbst in rep.tb:
                tb = tb + ’\n  File "%s", line %d, in %s’ % (
                    tbst[0], tbst[1], tbst[2])
                if tbst[3]:
                    tb = tb + "\n    %s" % (tbst[3],)
            debug("Msg recvrep exception: %s (%s)%s" % \
                  (‘rep.exc‘, rep.val, tb))
            raise rep.exc, rep.val
        return rep

    def message(self, req):
```

```
        R"""Send a message and receive a reply
        Sends a message to the node and port saved earlier and wait for a reply.
        """
        return(self.recvrep(self.sendreq(req)))


    def announce(self, req):
        R"""Send a message
        Sends a message to the node and port saved earlier.
        """
        self.sendreq(req).close()


class _Flow:
    R"""The base flow class
    The common part of the FlowSrc and FlowSink class. The flow classes use connected UDP!
    """
    def __init__(self, node = "", port=0):
        R"""Initialize a flow class
        Create a socket and save some information about the source or sink.
        """
        self.node = fetchnode(node)
        self.port = port
        self.open = 0
        self.flowsocket = socket(AF_INET, SOCK_DGRAM)

    def __del__(self):
        R"""Clean up
        Close the socket when done.
        """
        self.stop()
        if self.flowsocket:
            self.flowsocket.close()
        self.flowsocket = None

    def start(self):
        R"""Start the flow
        Packets will be transfered through the flow.
        """
        self.open = 1

    def stop(self):
        R"""Stop the flow
        Packet will be thrown away (ignored) in the flow.
        """
        self.open = 0


class FlowSrc(_Flow):
    R"""The source of a flow.
    The source of a flow implements a put method to send frames of the flow to the sink.
    """
```

```
    def __init__(self, node = "", port=0):
        R"""Initialize the source

        Prepare the source to send date to the sink.

        """
        _Flow.__init__(self, node, port)
        self.flowsocket.connect((self.node, self.port))

    def put(self, data):
        R"""Send the data to the sink

        Serialize and send the data to the sink of this flow.

        """
        if self.open:
            sendmsg(self.flowsocket, serialize(data))


class FlowSink(_Flow):
    R"""The sink of a flow

    The sink of a flow receives the frames (with size size) of the flow from the source.

    """

    def __init__(self, port=0):
        R"""Initialize the sink

        Prepare the sink to receive date from the source.

        """
        _Flow.__init__(self, "", port)
        self.flowsocket.bind((self.node, self.port))

    def get(self):
        R"""Receive the data from the source

        Receive and deserialize the data received from the source.

        """
        pklist = recvmsg(self.flowsocket)
        if self.open:
            pks = []
            for pk in pklist:
                pks.append(deserialize(pk))
            return pks
        else:
            return []


class _Event:
    R"""The base event class

    The common part of the EventSrc and EventSink class. The event classes use connected TCP/IP.

    """

    def __init__(self, node = "", port=0):
        R"""Initialize a flow class

        Create a socket and save some information about the source or sink.

        """
        self.node = fetchnode(node)
        self.port = port
        self.listensocket = socket(AF_INET, SOCK_STREAM)
```

```
    def __del__(self):
        R"""Clean up

        Close the socket when done.

        """
        self.stop()
        if self.listensocket:
            self.listensocket.close()
        self.listensocket = None


class EventSrc(_Event):
    R"""The source of an event.

    The source of an event implements a sendreq method to send events to the sink.

    """

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

        Prepare the source to send date to the sink.

        """
        _Event.__init__(self, node, port)
        self.listensocket.connect((self.node, self.port))

    def sendreq(self, data):
        R"""Send the data to the sink

        Serialize and send the data to the sink of this flow.

        """
        sendmsg(self.listensocket, pack(data))


class EventSink(_Event):
    R"""The sink of an event

    The sink of an event receives the events from the source.

    """

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

        Prepare the sink to receive date from the source.

        """
        _Event.__init__(self, node, port)
        self.listensocket.bind((self.node, self.port))
        self.listensocket.listen(1)
        self.notaccepted = 1

    def recvreq(self):
        R"""Receive the data from the source

        Receive and deserialize the data received from the source.

        """
        if self.notaccepted:
            self.listensocket_org = self.listensocket
            self.listensocket, addr = self.listensocket.accept()
            self.notaccepted = 0
            self.node, self.port = addr
        requests = []
        for req in recvmsg(self.listensocket):
```

```
            requests.append(unpack(req))
        return (self.listensocket, requests)


# LocalWords:  Aug Oct UK NORUT aacodefont defaultValues pythonrc py stderr AF
# LocalWords:  ErrorObject OpenORBException marshall unmarshall def init exc
# LocalWords:  msg BUFSIZ nsPort nmPort firstPort AttributeError str cPickle
# LocalWords:  marshalled unmarshalled misc MarshallException marshalling len
# LocalWords:  unmarshalling TupleType ListType DictType isinstance IRef obj
# LocalWords:  expID impID LBindCtrl iref CapsuleProxy ipnumexpr gethostbyname
# LocalWords:  listensocket INET sendreq req msgsocket recvreq addr tb recvrep
# LocalWords:  recv sendrep
```