

O₂S Components Framework: A HOWTO Guide

Day 2, Session 1
July 2006
Quanta JustPlay Seminar

Justin Mazzola Paluska
jmp@mit.edu

Acknowledgements

- These slides are derived from slides used in MIT class 6.894 as a tutorial on O2S.

Contents

- Definitions and System Architecture
- Concepts and Code
 - Synchronous
 - Asynchronous
 - Streams
 - Entity

What is an NPOP?

- NPOPs are:
 - real, in-process objects with methods and state
 - network objects that can be passed to remote hosts.
 - Remote hosts can invoke method calls on these resources.
- $\text{NPOP} = \text{RService} = \text{Resource} = \text{Service}$

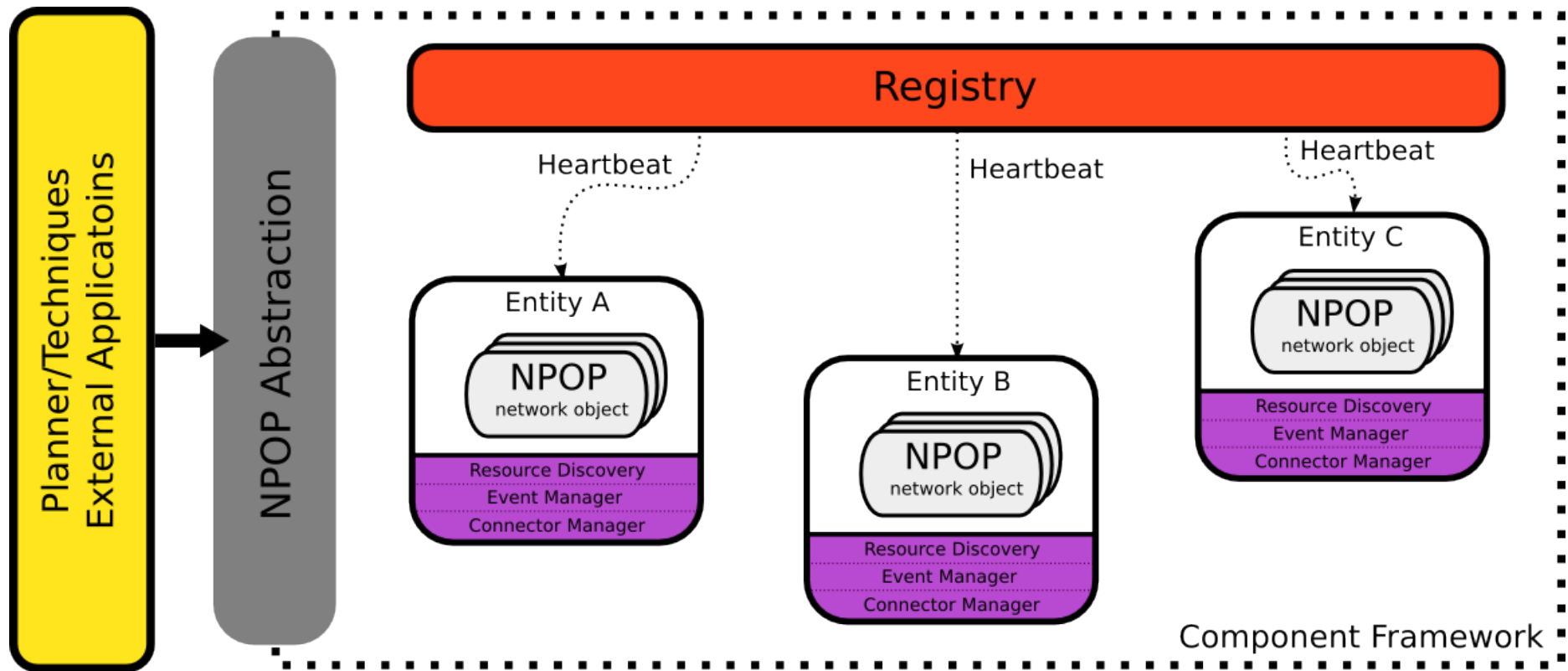
NPOP Model

- Remote Procedure Call
 - Server contains the actual NPOP object
 - Client contains stub objects that call back to the server
- Many 3rd-party RPC and stream packages from which to choose.
 - NPOPs use standard XML-RPC
 - NPOPs use TCP for streaming
 - with extensions for UDP, RTP, etc.

NPOP Model: Objectives

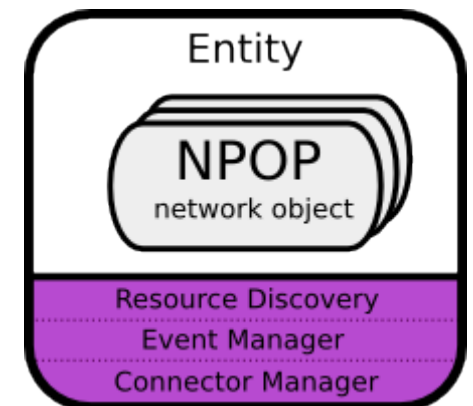
- Keep it simple:
 - for the NPOP developer
 - for the NPOP user
 - for every one!
- Support many Platforms and Types
 - Language independent Types and Marshaling

Scope: O2S Components



Entity

- Entities:
 - register with the Registry
 - are health-monitored by the Registry
 - Through a UDP keepalive “heartbeat”
 - manage 1 or more NPOPs
 - acts as the “gateway” to from the Registry to NPOPs



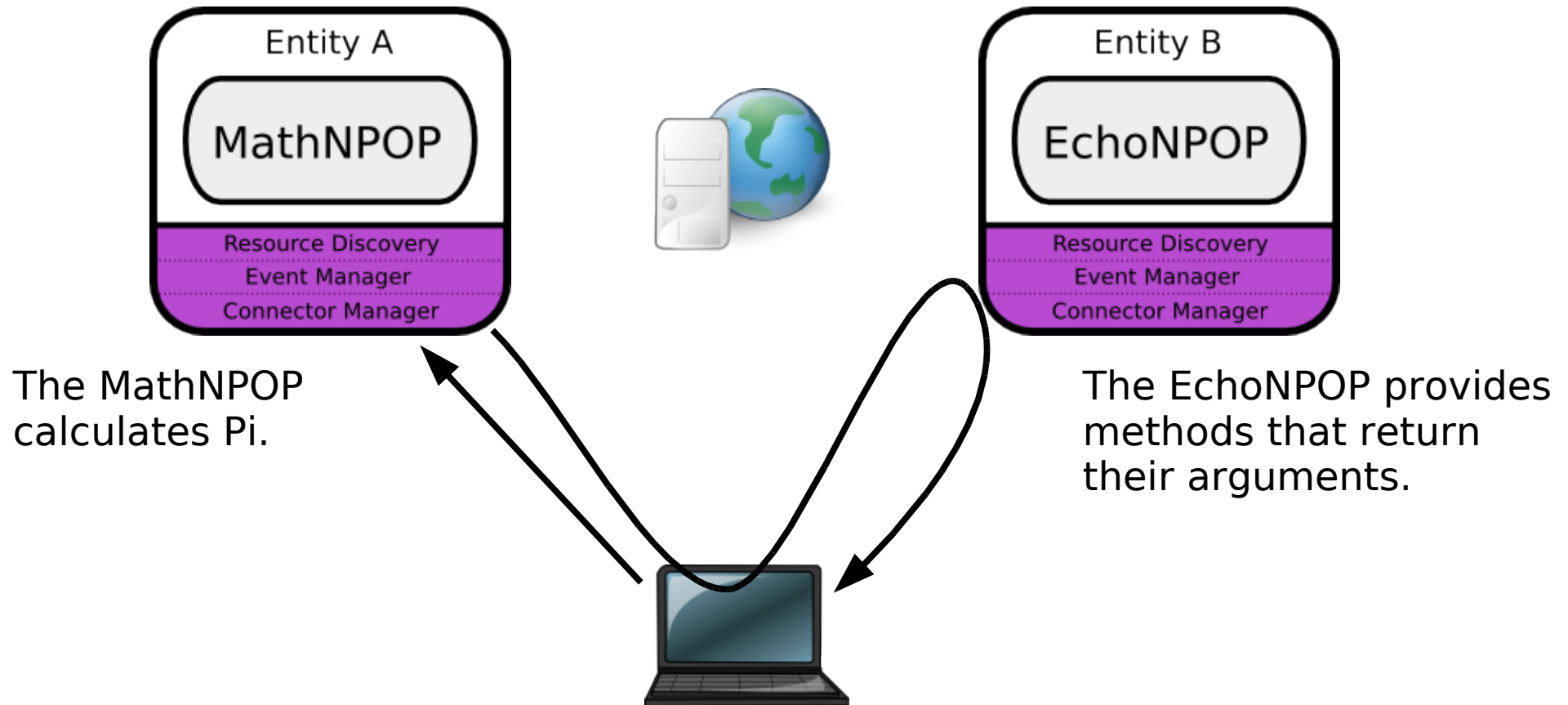
Entity

- In other words, the Entity represents a logical host in O2S
- Entities provide a few O2S-specific services to its managed NPOPs:
 - Event Handling
 - Connectors
 - Registry Services
- It is possible to use an NPOP outside of an Entity, but you lose access to the O2S-specific services.

Concepts in Code

- Simple Synchronous Example
 - What can be passed between Hosts?
 - Local Versus Remote Calls
 - Anatomy of an NPOP
- Asynchronous Notifications
- Streams
- Entity
 - Bootstrapping

Simple Example



Assume that the laptop client has a handle to the MathNPOP on Entity A called **math_npop** and a handle to the EchoNPOP on Entity B called **echo_npop**.

Simple Example

```
class MathNPOP(RService):  
    ...  
    def o2s_get_pi(self):  
        pi = self.compute_pi()  
        return pi
```



```
class EchoNPOP(RService):  
    ...  
    def o2s_echo(self,  
                  msg,  
                  arg=None):  
        print msg  
        if not arg:  
            return self  
        else:  
            return arg
```



```
# assume we have instances of MathNPOP and EchoNPOP  
# named math_npop and echo_npop at t0  
  
rval_npop = echo_npop.echo('Hi!', arg=math_npop)  
print rval_npop == math_npop # True, same object in memory  
print rval_npop.get_pi() # 3.1415926  
  
# EchoNPOP will send itself back if no kwargs given!  
rval_npop2 = echo_npop.echo('Hi again!')  
print rval_npop2 == echo_npop # True, same object in memory
```

What can be passed?

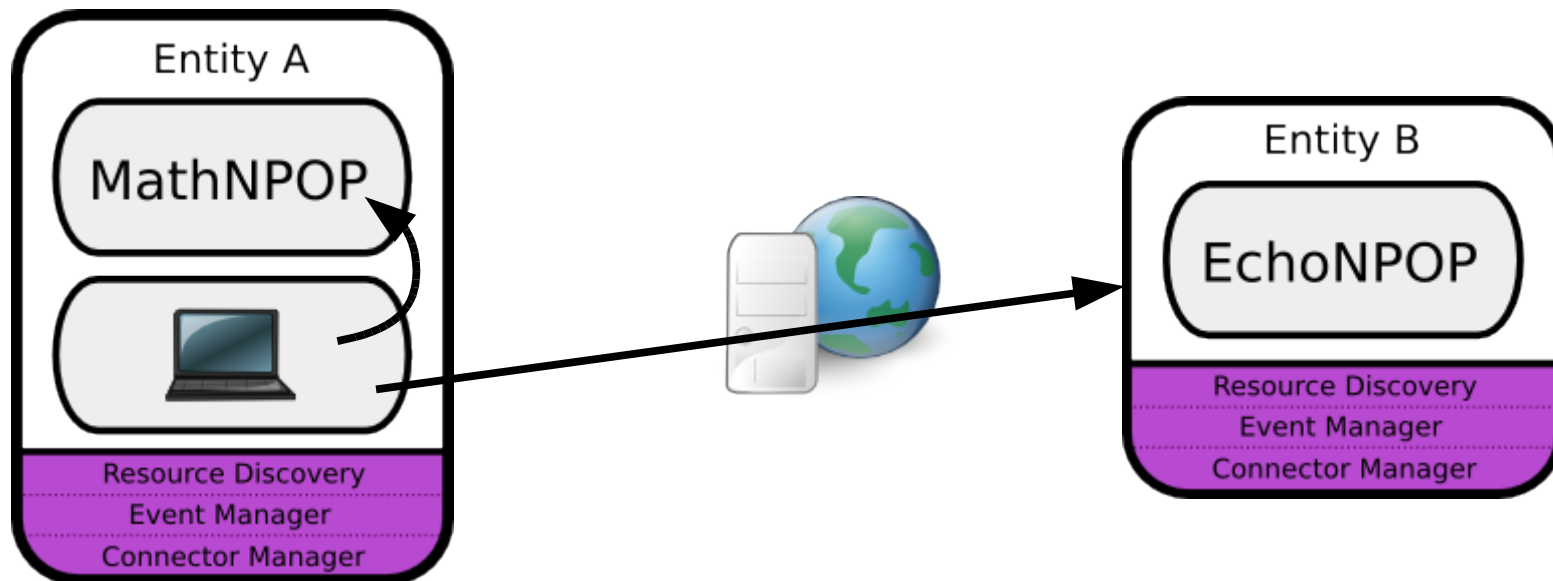
- NPOPs will transparently pass subtypes of the RType class
 - integers, floats, strings
 - None
 - Lists, Dictionaries
 - RServices (the NPOP class)
- RService sub-classes are automatically marshaled into stubs as necessary

Properties of RTypes

- RTypes are pass-by-copy when passing through a process boundary
 - Uses underlying language semantics if passing within a process.
- RServices are pass-by-reference
 - Remote hosts get RStubs
 - == works as expected

Local versus Remote Calls

- The NPOP user's code can be anywhere
 - The location of an NPOP can be determined at runtime.
- What if code the user's code is in Entity A?



Local versus Remote Calls

- User's code still works – but the method calls on MathNPOP are now local, in-process calls.
 - No network traffic or marshaling overhead for local calls!



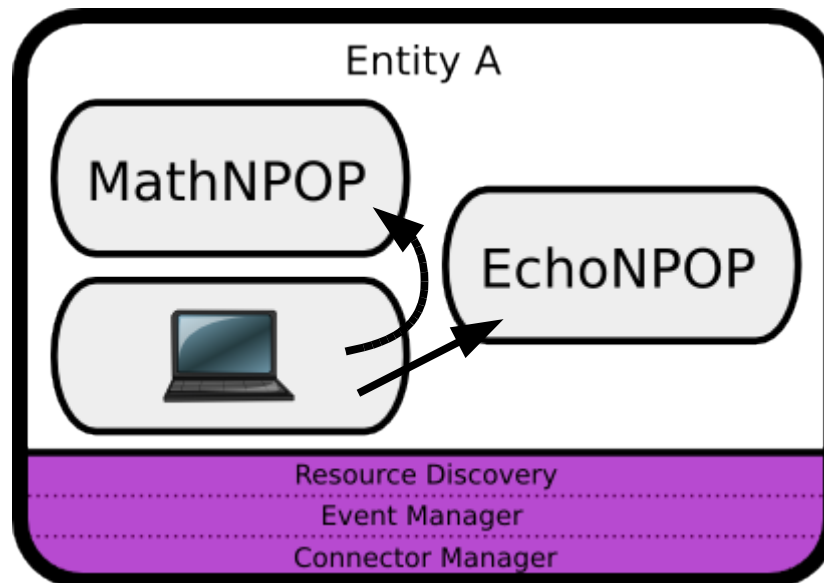
```
# assume we have instances of MathNPOP and EchoNPOP
# named math_npop and echo_npop at t0

rval_npop = echo_npop.echo('Hi!', arg=math_npop)
print rval_npop == math_npop # True, same object in memory
print rval_npop.get_pi() # 3.1415926

# EchoNPOP will send itself back if no kwargs given!
rval_npop2 = echo_npop.echo('Hi again!')
print rval_npop2 == echo_npop # True, same object in memory
```

Local versus Remote Calls

- If MathNPOP, EchoNPOP, and the user's code live on the same Entity, then all method calls are local.
 - Zero network traffic



Local versus Remote Calls

- In other words...
 - Users of NPOPs should not know (or need to know) whether the code implementing an NPOP is local or remote.
 - The system automatically does the right thing (e.g., local calls for local objects)
 - Unlike other architectures, NPOPs do not expose user-level stubs

Making NPOPs

```
class MathNPOP(RService):  
  
    def __init__(self, entity, **kwargs):  
        RService.__init__(self, entity)  
        # ...  
  
    def o2s_get_pi(self):  
        return self.compute_pi()  
  
    def compute_pi(self):  
        # crunch numbers...  
        return 3.1415926  
  
    def o2s_get_e(self):  
        raise NotImplementedError()  
  
    def o2s_many_args(self, arg1, arg2,  
                      *args, **kwargs):  
        # in Python, all forms of  
        # arguments are allowed...  
  
        # ...and you can pass None  
        return None  
  
    def o2s_return_self(self):  
        return self
```

Subclass RService (from the system.rservice package)

Pre-pend o2s_ to methods that should be remotely accessible.
Users do not need to use o2s_ in their function calls.

Remotely accessible methods in NPOPs can raise Exceptions like normal code.

All types of arguments are supported.

If you need to pass yourself to other NPOPs, just return self

Exceptions

- NPOPs support Exceptions
 - If a remote NPOP causes an exception, the exception will be marshaled and raised on the local side.
 - The system will also try to make a local traceback.

```
Error: Exception Thrown on Remote Host.  
Could not handle `get_e` request to resource at host:  
( '192.168.0.117', 55567)  
* REMOTE TRACEBACK:  
system.exception.ResourceError:Error on: 192.168.0.117:55567  
Traceback (most recent call last):  
  File "/home/jmp/Work/o2s/system/listener.py", line 146, in _dispatch  
    result = rservice._dispatch(method, *args, **kwargs)  
  File "/home/jmp/Work/o2s/system/rservice.py", line 341, in _dispatch  
    return method_ptr(*params, **kwargs)  
  File "math_npop.py", line 19, in o2s_get_e  
    raise NotImplementedError()
```

Concepts in Code

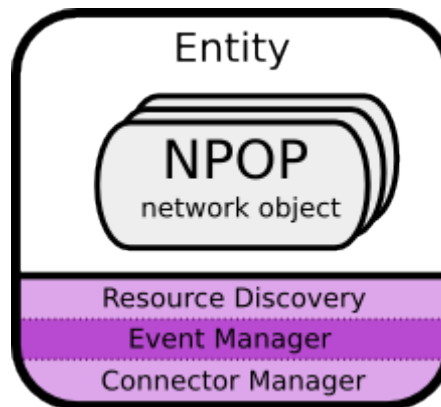
- Simple Synchronous Example
 - What can be passed between Hosts?
 - Local Versus Remote Calls
 - Anatomy of an NPOP
- Asynchronous Notifications
- Streams
- Entity
 - Bootstrapping

Beyond Synchronous Calls

- RPC works well for synchronous function calls – but isn't well suited for other kinds of calls
- Asynchronous Callbacks and notifications?
O2S Events
- Fast, stream-oriented connections between NPOPs?
O2S Stream Connectors

O2S Events

- Two key concepts (and classes):
 - Events – RType objects
 - Event Listeners



Events

- Events are asynchronous notifications (with an arbitrary payload).
- Events are RTypes
 - Have method calls – mostly “getters”
 - Passed by copy to remote hosts.
- Events are sent between NPOPs
 - Usually between EventListener NPOPs

Event Listeners

- NPOPs that receive dispatched Events
- The Entity class provides NPOPs with Event Listeners
- Event Listeners are NPOPs:
 - have methods (“throw_event”)
 - can be passed to remote hosts

Events Example 1

- Entity manages the Event queue:

```
class ServerNPOP(RService):  
  
    def o2s_get_el(self):  
        entity = self.get_entity()  
        return entity.get_event_listener(self.handle_event)  
  
    def handle_event(self, event):  
        if event.get_data() == "EVACUATE":  
            print "Emergency! %s" % event.get_message_string()
```



```
el = npop.get_el()  
event = Event(message_type="WARNING",  
               thrower=self, # if self is an NPOP  
               recipient="the usual suspects",  
               message_string="evacuate now!",  
               data='EVACUATE',  
               parameters={'time' : 'NOW'})  
# actually throw the event  
el.throw_event(event)
```

Events Example 2

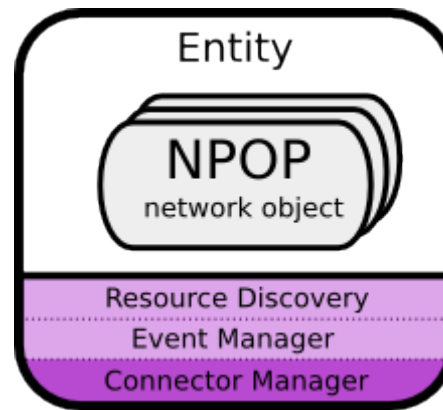
- Programmer manages the Event queue:

```
class ServerNPOP(RService):  
  
    def o2s_get_el(self):  
        entity = self.get_entity()  
        # NOTE: don't provide a target...  
        self.el = entity.get_event_listener()  
        th = threading.Thread(target=self.handle_event)  
        th.setDaemon(True)  
        th.start()  
        return self.el  
  
    def handle_event(self):  
        while True:  
            # get_event is a blocking call  
            event = self.el.get_event()  
            if event.get_data() == "EVACUATE":  
                print "Emergency! %s" % event.get_message_string()
```



Stream Connectors

- Entity provides NPOPs with a Connector Manager
 - creates and manages stream connections
 - analogous to the Event / EventListener framework



Stream Connectors

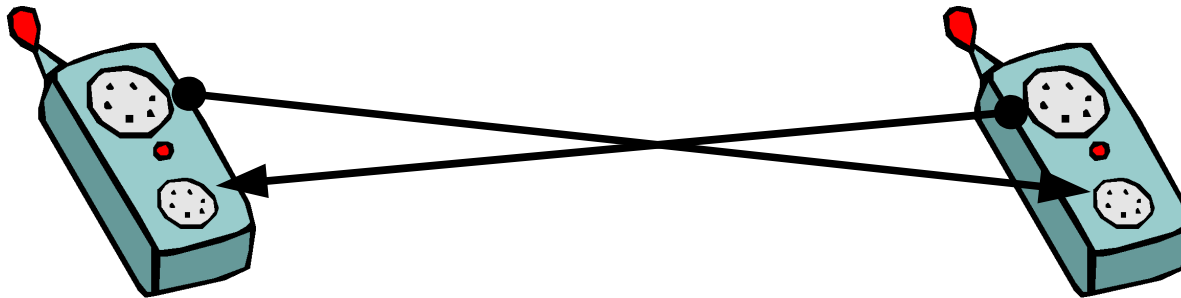
- Connectors are fast, stream-oriented connections between NPOPs
 - Unidirectional
 - Raw bytes
- Connectors are also NPOPs
 - can be passed between NPOPs
 - RPC calls are used only to open, close, and control the stream connections.

Connectors Example (Client)

- Connect two “walkie talkies” together

```
(in1, out1) = walkie1.get_connectors()
(in2, out2) = walkie2.get_connectors()

Connect(in1, out2)
Connect(in2, out1)
```



Connectors Example (Server)

```
class ServerNPOP(RService):

    def __init__(self, entity):
        RService.__init__(self, entity)
        self._make_connectors()

    def _make_connectors(self):

        entity = self.get_entity()
        self.in_con = entity.make_connector(name="Audio In", direction='in',
                                            handler=self.handle_incoming_data)
        self.out_con = entity.make_connector(name="Audio Out", direction='out')

    def o2s_get_connectors(self):
        # Connectors are NPOPs!
        return (self.in_con, self.out_con)

    def handle_incoming_data(self, data,
                             connector_npop, meta):
        if data == None:
            # connector closed, so remove the connector
            del self.in_con
        else:
            pass # ...play audio...

    def push_outgoing_audio(self, sound_data):
        try:
            self.out_con.send(sound_data)
        except ConnectorError:
            # not connected, or connection broken,
            # so remove the connector
            del self.out_con
```

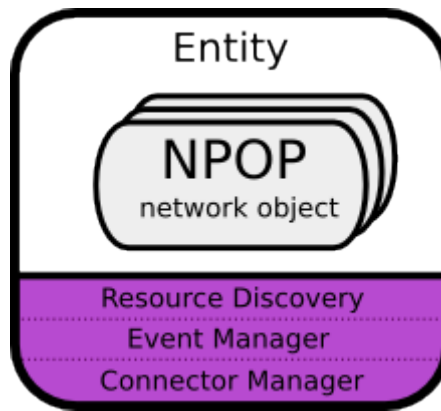


Concepts in Code

- Simple Synchronous Example
 - What can be passed between Hosts?
 - Local Versus Remote Calls
 - Anatomy of an NPOP
- Asynchronous Notifications
- Streams
- Entity
 - Bootstrapping

The Entity

- Container class that organizes, starts, and manages NPOPs



Entity

- An Entity represents a Logical Host
- By default, the entity provides:
 - NPOP Management (sending and receiving NPOPS)
 - Event Listening/Dispatching
 - Connector Management
 - Registry Services
- The Entity class is an NPOP

Using the Entity

- To execute NPOPs, extend the Entity class and mix-in your NPOPs
 - Subclass Entity and contain NPOPs
 - Use Multiple Inheritance
- JustPlay demo uses a combination of both methods.

Extending Entity

- NPOP Containment
 - Must provide a way to access the internal NPOPs

```
class MathEntity(Entity):  
    def __init__(self):  
        Entity.__init__(self)  
        self.math_npop = MathNPOP(self)  
  
    def o2s_get_pi(self):  
        return self.math_npop.compute_pi()  
  
    def o2s_get_math(self):  
        return self.math_npop
```

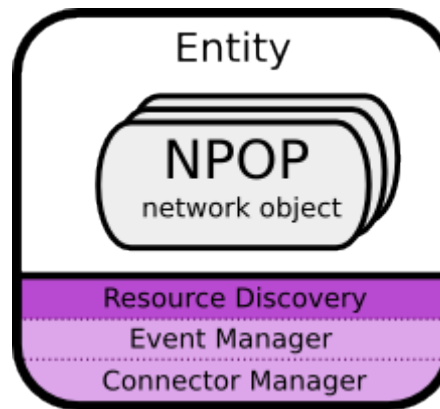
Extending Entity

- Multiple Inheritance
 - Automatically exposes all methods of the NPOPs
 - Can be harder to understand sometimes

```
class MathEntity(Entity,  
                  MathNPOP):  
  
    def __init__(self):  
        Entity.__init__(self)  
        MathNPOP.__init__(self, self)
```

Registry Services

- The Entity provides a link to the O2S Registry
- The Registry provides:
 - discovery services
 - health monitoring services



Registry Services: Bootstrapping

- So far, we've assumed that the NPOP user has had access to a few NPOPs at the start.
- We know it's possible to pass NPOPs between hosts.
- How did the remote hosts get their NPOPs to begin with?

Bootstrapping

- The Entity provides a “find_entities” method that looks up Entities in the Registry and returns a handle to them.

```
class ServerEntity(Entity):  
    def __init__(self):  
        Entity.__init__(self)  
  
        # find_entities returns a list of dictionaries. The  
        # dictionaries represent everything the Registry knows about  
        # the Entities.  
        walkies = self.find_entities(entity_type = "Walkie",  
                                     health = "Healthy",  
                                     name = self.name)  
  
        # pick one (NOTE: be careful here, there may be 0!)  
        walkie = walkies[0]  
        # extract NPOPs from Registry keys  
        walkie_npop = walkie['resource']  
  
        print "walkie_npop:", walkie_npop
```

Health Monitoring

- The Entity also provides the “registry_subscribe” method to be notified about the health of other Entities.
- Commonly used for:
 - bootstrapping – the Registry informs the NPOP when new Entities are available
 - adaptivity – react to changes in NPOP availability

Registry Subscriptions

```
class ServerEntity(Entity):  
    def __init__(self):  
        Entity.__init__(self)  
  
        self.el = self.get_event_listener(self.handle_notification)  
        self.sub_id = self.registry_subscribe(listen_for_event='updown',  
                                              event_listener=self.el,  
                                              entity_type="Walkie")  
  
    def handle_notification(self,  
                           event):  
  
        print "Notification:", event
```

Contents

- Definitions and System Architecture
- Concepts and Code
 - Synchronous
 - Asynchronous
 - Streams
 - Entity
- Custom Network Objects
- “Feature”

Custom Network Objects

- The built-in set of RTypes provides almost everything you'll need
 - most standard language primitives
 - NPOPs
- The built-in set of RTypes is automatically converted to the wire types of XMLRPC
 - language and platform independent

Custom Network Objects

- In O2S, RTypes and custom NPOPs should almost always suffice.
- Benefits of using NPOPs
 - Interning
 - Efficient multiplexing of a single XML RPC server
 - Language and Platform independence
 - An NPOP written in Python will automatically work on a remote Java (or any other supported language) host.

Custom Network Objects

- Determining if you really need to make your own non-NPOP object:
 - The remote Entity **must** have access to the module with your custom class in order to unmarshal your custom object
 - You must port your custom class to all platforms.
- RTypes are extensible

Custom Network Objects

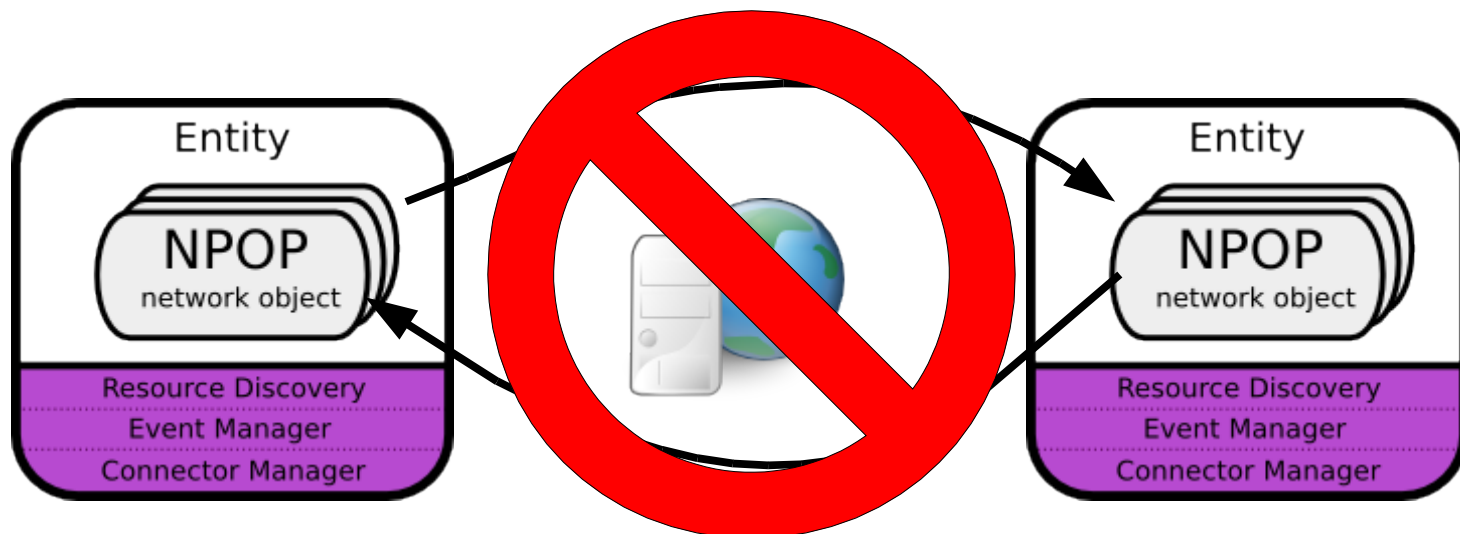
- Example: Events are custom RTypes
 - NPOP model doesn't make sense
 - Events have methods, but Event objects don't fit the client/server model
- See `system/event.py` for more details

Contents

- Definitions and System Architecture
- Concepts and Code
 - Synchronous
 - Asynchronous
 - Streams
 - Entity
- Custom Network Objects
- “Feature”

Feature? Bug?

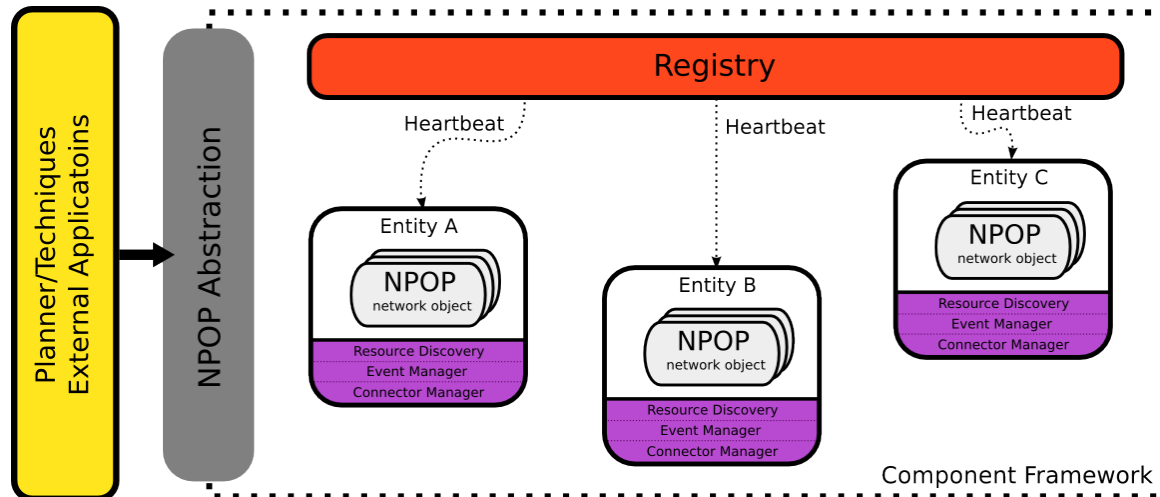
- We prefer “open research question”
- The NPOPs framework presents a single-threaded view of a distributed, parallel world.
 - “request loops” lead to deadlock



Contents

- Definitions and System Architecture
- Concepts and Code
 - Synchronous – NPOPs
 - Asynchronous – Events
 - Streams – Connectors
 - Entity – Registry

Questions?



Day 1, Session 3
July 2006
Quanta JustPlay Seminar

Justin Mazzola Paluska
jmp@mit.edu