

distributed programming with py.execnet

Holger Krekel
<http://merlinux.eu>

EuroPython 2009, Birmingham

July 2nd, 2009



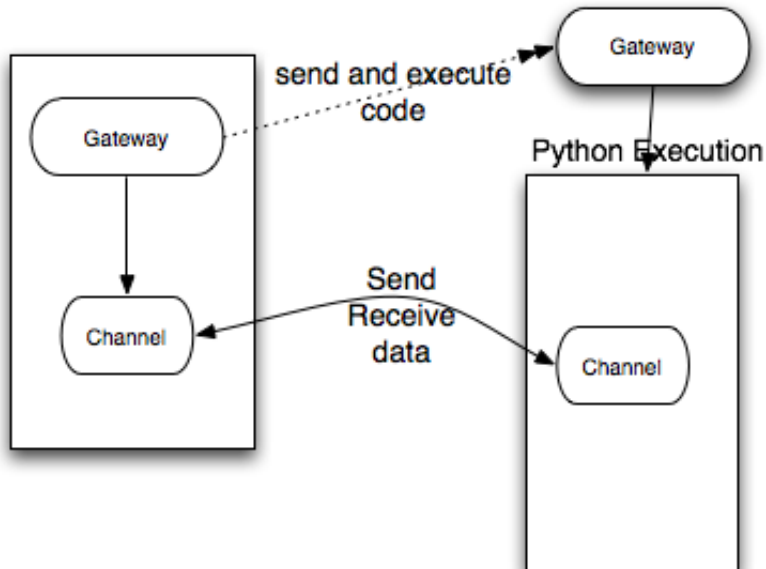
my distributed systems background

- programming since 20 years, Python since around 2000
- Adaptive Communication Environment (ACE)
- Corba ORB TAO, C++ Corba Transaction Service (XOTS)
- automated remote system maintenance
- py.test distribution of tests
- consultancies on RMI / distributed (C++/Java) systems

This talk

- intro to py.execnet, examples, demos
- some comparison notes (multiprocessing / RMI systems)
- summary / future bits

The basic model / terminology



What execnet does

1. instantiate remote Python process
2. send code for remote execution
3. send/receive of data between remote and local code

elastic code: gateway's remote_exec

```
>>> gw = py.execnet.PopenGateway()
>>> src = "import os ; channel.send(os.getpid())"
>>> channel = gw.remote_exec(src)
>>> remote_pid = channel.getpid()
```

What about remote failures?

`channel.RemoteError` textually represents remote traceback

Channel send/receive

- channels have symmetric send/receive
- data items are Python primitive types (dict, int, string, ...)

Gateways

- gateways are represented on both sides
- each gateway object spawns a receiver thread
- but execution is single-threaded by default

Gateway / process instantiation

currently three gateway types:

- PopenGateway opens a local Subprocess
- SshGateway opens a ssh-mediated remote subprocess
- SocketGateway opens a socket-mediated remote subprocess

Gateways work **across Python versions**, for example: a Python-2.4 process can connect with a Python-2.6 subprocess

SshGateway zero-install deployment

- uses underlying “ssh” command line tool
- only requires plain Python Interpreter
- no prior installation of remote code
- sends code *elastically*

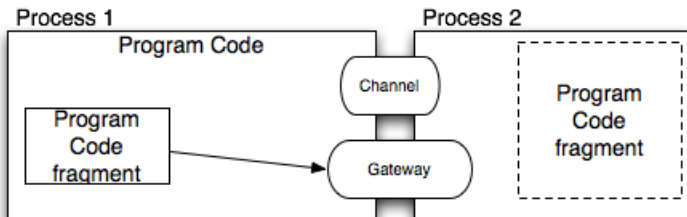
example: getting remote machine info

- single-file script to gather info from remote host

example: svn-hotsyncing repositories

- single-file script to hot-sync svn repository

“elastic code”



Some helpful “on-top” API



<http://marikaz.deviantart.com/> CC 3.0 AN-ND

Channel objects, waitclose()

waiting for creation of a file:

```
>>> ch = gw.remote_exec("""
    filename, content = channel.receive()
    f = open(filename, 'w')
    f.write(content)
    f.close()
    """) # channel.close() is called automatically
>>> ch.send("hello.txt", "world")
>>> ch.waitclose()
```

receive data from multiple sub processes

Use MultiChannels for receiving multiple results from remote code:

```
>>> from py.execnet import PopenGateway, MultiChannel
>>> src = "import os ; channel.send(os.getpid())"
>>> ch1 = PopenGateway().remote_exec(src)
>>> ch2 = PopenGateway().remote_exec(src)
>>> mch = MultiChannel([ch1, ch2])
>>> l = mch.receive_each()
>>> assert len(l) == 2
```

async-receive data from sub processes

Use `MultiChannel.make_receive_queue()` for asynchronously receiving data from remote code. The returned queue yield `(channel, result)` tuples allowing to determine where a result comes from:

```
>>> ch1 = PopenGateway().remote_exec("channel.send(1)")
>>> ch2 = PopenGateway().remote_exec("channel.send(2)")
>>> mch = MultiChannel([ch1, ch2])
>>> queue = mch.make_receive_queue()
>>> chan1, res1 = queue.get() # optional timeout
>>> chan2, res2 = queue.get()
>>> res1 + res2
3
```

remote_exec is single threaded by default

```
ch1 = gw.remote_exec("channel.send(channel.receive()+1)")
ch2 = gw.remote_exec("channel.send(channel.receive()+1)")
ch2.send(1)
ch2.receive() # XXX BLOCKS XXX
```

concurrent remote execution

```
gw.remote_init_threads(num=2)
ch1 = gw.remote_exec("channel.send(channel.receive()+1)")
ch2 = gw.remote_exec("channel.send(channel.receive()+1)")
ch2.send(1)
ch2.receive() # now works!
ch1.send(3)
ch1.receive()

[quick look into remote_init_threads implementation]
```

XSpecs and `py.execnet.makegateway()`

1.0 introduces a “XSpec” string-scheme for specifying gateways:

```
popen//python=2.5//nice=20
```

```
socket=192.168.1.4:8888
```

```
ssh=wyvern//python=python2.4//chdir=mycache
```

so you can do:

```
>>> gateway = py.execnet.makegateway(xspec)
```

rsyncing dependencies

use `py.execnet.RSync` to transfer packages:

```
>>> rsyncer = py.execnet.RSync("sourcedir")
>>> gw = py.execnet.SshGateway("codespeak.net")
>>> rsyncer.addtarget(gw)
>>> # add multiple targets
>>> rsyncer.send()
```

example: py.test distributed testing

```
py.test --dist=each \  
  --tx ssh=wyvern \  
  --tx ssh=code \  
  --rsync mypkg
```

multiprocessing (Python 2.6)

- aims for transparent thread/process distinction
- requires **pickling** to work for your app
- Pipes for one-to-one and Queues for many-to-many communication
- requires compatible “multiprocessing” package on the “remote” side
- requires to-be-executed code to **pre-exist**
- Python version agnostic? there are backported multiprocessing packages

Remote Method Invocation

- use object identity models
- static code distribution and static server code
- usually works with pickling
- pre-existing code on both sides

the “pre-existing” code requirement ...

- version-dependencies
- manual installation
- difficult to extend to “arbitrary” hosts
- difficult to automate tests
- **not developer friendly**

Execnet Summary

- ad-hoc instantiate remote Python processes
- send and execute code **elastically**
- zero-install - code is determined client-side
- communicate through symmetric channels
- works across host and Python interpreter barriers

Open issues

- improved debugging / tested teardown mechanisms
- channels/gateways to cross multiple hosts
- best practise model for sharing data
- **user contributed VMs for elastic execution**

holger @ merlinux.eu