The PicklingTools Library: A Toolkit for Combining C++ and Python

by Richard T. Saunders

for PyCon 2010 in Atlanta

A Motivating History and Tutorial

What are the PicklingTools?

- The PicklingTools are an Open Source library of Python code and C++ code
  - allows developers to build systems out of C++ parts and Python parts, and have those parts communicate
  - or
  - A collection of socket and file tools to allow C++ and Python to exchange Python Dictionaries

Philosophy: Python Dictionaries are Currency of the PicklingTools

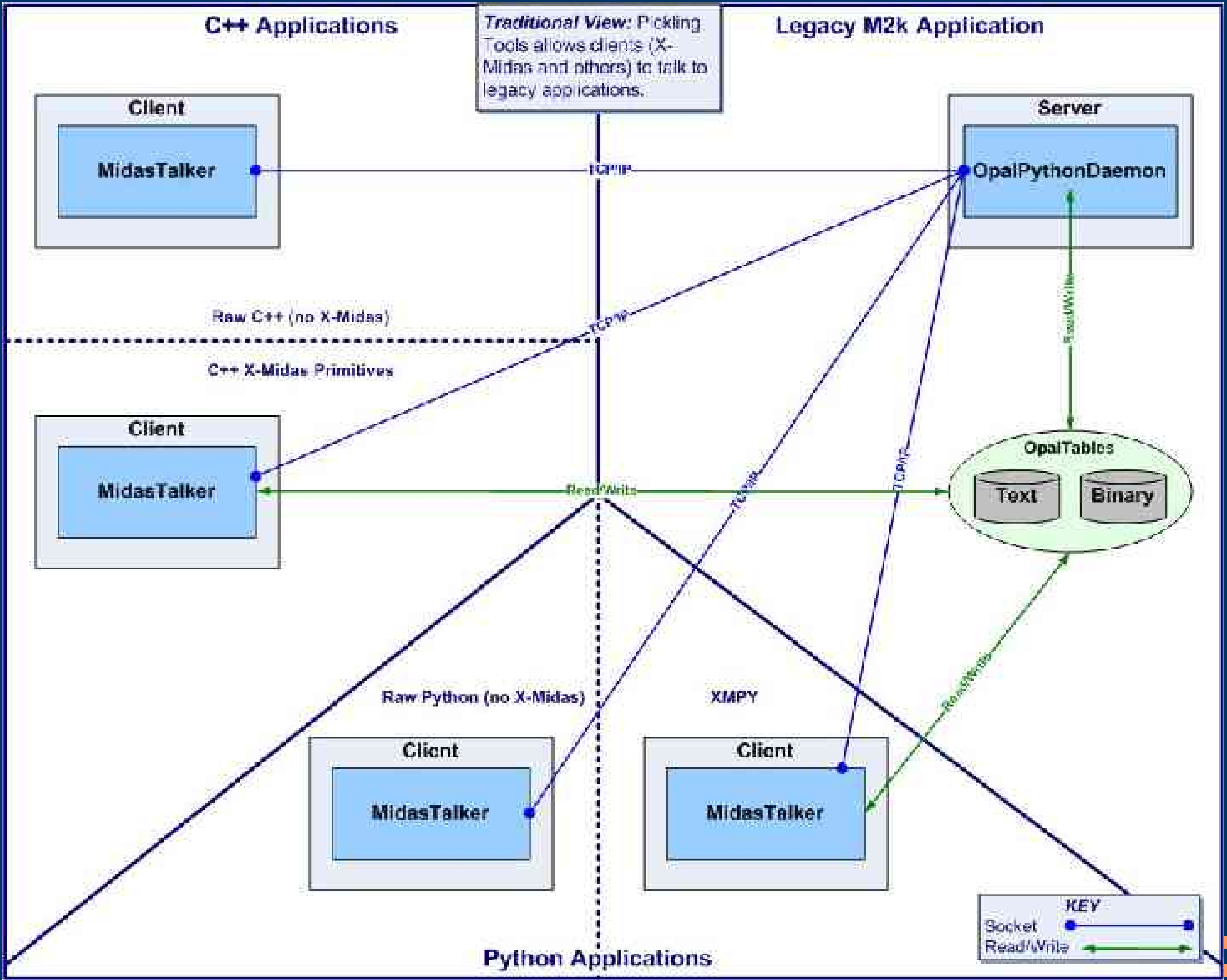
- All interactions between C++ and Python are via ***Python Dictionaries***
  - `{'retries':100,'request':'ping','time':5.5}`
  - Python Dictionaries stored in files, can read/write from either Python or C++
  - Python Dictionaries flow across sockets, can read/write from either Python or C++
- The toolset is called the PicklingTools because when Python Dictionaries are serialized, they are said to be *pickled*

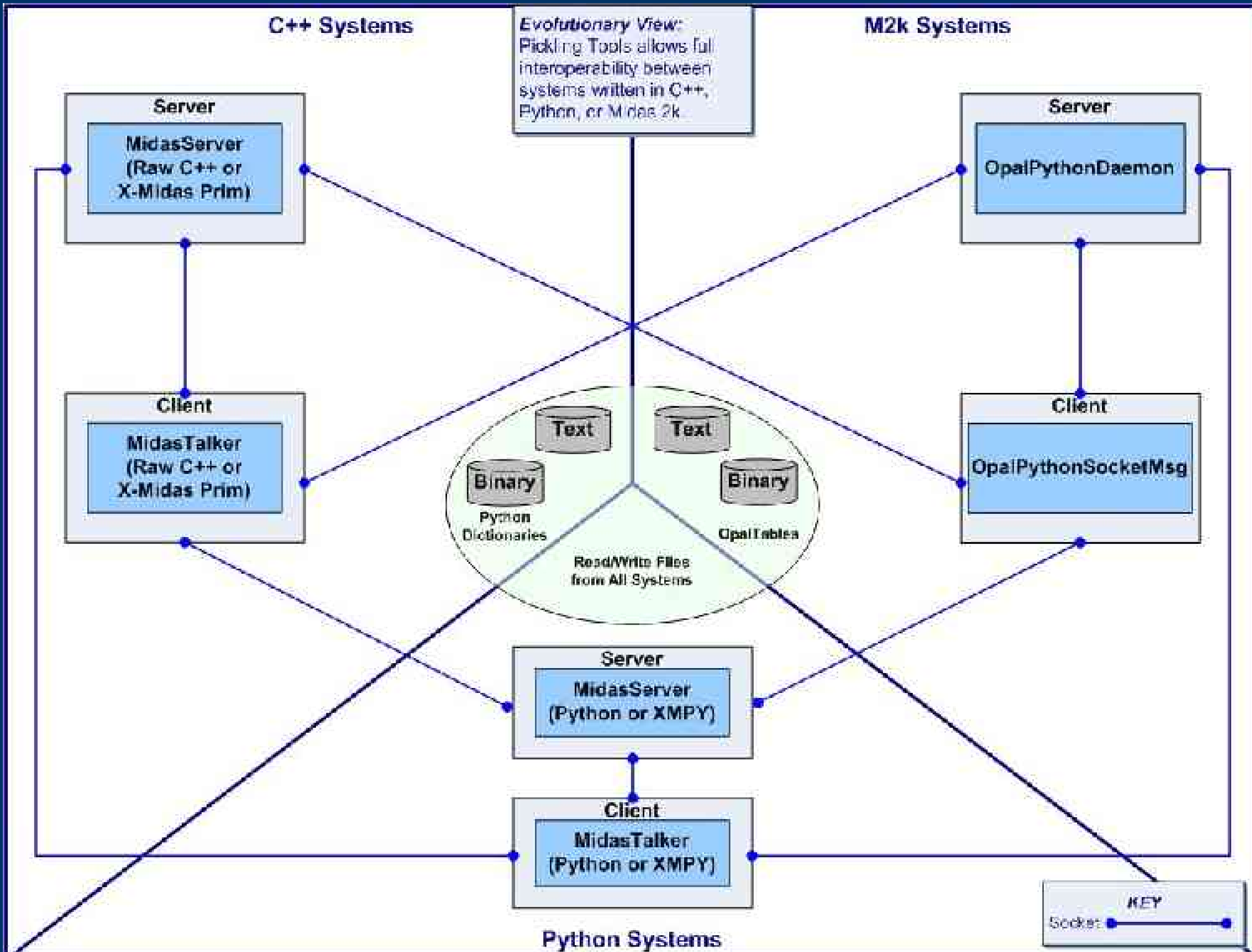
Overview of Tools in PicklingTools

- TCP/IP Servers and Clients: C++ and Python
  - called (resp.) *MidasServer* and *MidasTalker*
  - you DO NOT need Midas (name is historic remnant)
  - ... but CAN communicate with legacy Midas if need to
- UDP Servers and clients: C++ and Python
  - called (resp.) *MidasYeller* and *MidasListener*
  - again, you DO NOT need Midas (names historic)
- Read files w/many represent.: C++ and Python
  - Textual and Binary Serialized Python Dictionaries

Overview of Implementation of PicklingTools

- Python code: just wrappers to built-ins (no extension modules, just raw Python!)
  - Python Dictionaries: built-in, easy to manipulate
  - Socket code: import socket
  - Serialization code: import cPickle
- C++ code: goal is to feel like the Python side
  - Python Dictionaries: emulated though OpenContainers
  - Socket code: available on UN*X systems
  - Serialization code: reverse engineered Python Pickling Protocol 0 (7-bit clean) and 2 (binary), also text tables


C++ Applications
Traditional View: Pickling Tools allows clients (X-Midas and others) to talk to legacy applications.
Legacy M2k Application
Client
MidasTalker
TCP/IP
Server
OpalPythonDaemon
Raw C++ (no X-Midas)
TCP/IP
C++ X-Midas Primitives
Client
MidasTalker
Read/Write
OpalTables
Text
Binary
TCP/IP
TCP/IP
Read/Write
Raw Python (no X-Midas)
XMPY
Read/Write
Client
MidasTalker
Client
MidasTalker
KEY
Socket
Read/Write
Python Applications


C++ Systems
Evolutionary View:
PickIing Tools allows full interoperability between systems written in C++, Python, or Midas 2k
M2k Systems
Server
MidasServer (Raw C++ or X-Midas Prim)
Server
OpalPythonDaemon
Client
MidasTalker (Raw C++ or X-Midas Prim)
Text
Text
Binary
Binary
Python Dictionaries
OpalTables
Read/Write Files from All Systems
Client
OpalPythonSocketMsg
Server
MidasServer (Python or XMPY)
Client
MidasTalker (Python or XMPY)
KEY
Socket:
Python Systems

Legacy Systems use PicklingTools

- 8 years of my life.. summed up in one slide
  - GALACTUS: thousands of installs
    - one install uses entire machine
  - SILVER SURFER: 378000 lines of Python/C++ code
    - runs on 400+ quad-code machines
  - NOVA: 406000 lines of Python/C++ code
    - runs on 120+ quad-core machine
  - see paper on history on web site

Spring 2008: Software Engineering class at the University of Arizona

- FULL CLASS PROJECT
- Arkham Horror:
  - complex table-top game
    - complicated rule-set
  - 100s of cards, pieces
    - each card subtly changes the rules of the game
- Cries out for computerization
  - networking ... so everyone doesn't have to sit at table
  - have computer handle rules, upkeep

Arkham Horror Architecture

- Model-View-Controller
  - Game engine handles and keeps all state
    - player locations, health, monster locations, etc.
    - implemented as a `MidasServer`
  - Players sit at separate computers, play over network
    - Client shows current state of the board
    - implemented as a `MidasTalker`

Rules Rules Rules

- So many cards, so many subtle rules ...
- Game Engine is a “Prolog-like” game engine
  - rules encoded as Python Dictionaries
  - each card is a set of rules
    - card processed by engine when card is “revealed”
  - avoids hard-coding all logic in game
    - game is in the cards
  - makes it easy to add “expansions” (currently 6)
    - just add new bunch of tables with the new “rules”

Sample Cards

Sample Encoding (pretty print)

```
{
  'Monster' : 'The One Who Cannot Be Named',
  'Attributes': [
     'Physical Resistance',
     'Magical Resistance'
  ],
  'Defense': -3,
  'DoomTrack': 14,
  'Attack' : {
     'Will':+2,
     'Frequency': ['perturn', -1]
  },
  'Picture' : 'unnamed.jpg'
}
```

A MidasTalker is the Client

```
from midastalker import *   # raw Python
mt = MidasTalker(“dl380”, 8888, SERIALIZE_P0)

mt.open()      # Connect!

# Send a request to server
request = { 'PING': { 'timeout': 5.0 } }
mt.send(request)

# Receive response back, wait up-to 4 seconds
response = mt.recv(4.0)   # Returns None if
                          # no response in 4

if response==None : error_out()
```

A MidasTalker is the Client (Adv.)

```
from midastalker import *
mt = MidasTalker(“dl380”, 8888, SERIALIZE_P0)

while 1:
  try :
    mt.open()  # Can we connect?
  except :
    print '... retry to connect in 5 seconds'
    time.sleep(5)

while 1:
  try :
    request = CreateRequest() # Some user fun
    mt.send(request)
  except :
    print ' ... server went away?  Retry to connect'
```

MidasTalker examples

- Examples of how to use the MidasTalker litter the baseline:
  - PicklingTools104/Python/
    - midastalker_ex.py   # easy to read and understand, fragile
    - midastalker_ex2.py # harder to read, robust with error hand.
  - PicklingTools104/C++/midastalker_ex.cc
    - midastalker_ex.cc   # as above, easy but fragile
    - midastalker_ex2.cc # as above, hard but robust
  - PicklingTools104/Xm/ptools104/host
    - xmclient.cc   # How to use in X-Midas framework

Problem: How do we emulate Dynamic Types in C++?

- Consider in Python (dynamically typed language):

```
a = 1
b = "hello"
a = b          # okay
```

C++ statically typed: types known at compile time

```
int a = 1;
string b = "hello";
a = b; // Compiler error! Different types!
```

Solution: Use Val to represent Dynamic Types in C++

- Val is the C++ type that means “dynamic typing”
- C++ Val: heterogeneous container of any basic type in C++

```
Val a = 1;          // int
Val b = “hello”;    // string
a = b; // Okay, a & b same type
```

Val is essentially a union type for all basic C++ types: int_1, int_u1, int_2, int_u2, int_4, int_u4, int_8, int_u8 real_4, real_8, complex_8, complex_16, None, Tab

Val Constructor for all Basic Types

```
Val v1 = “hello”;              // string
Val v2 = 1.0;                  // real_8
Val v3 = 1.0f;                 // real_4
Val v4 = complex_8(1,2); // complex
Val v5 = None;                 // none
Val v6 = Tab();       // empty table
Val v7 = 17;                   // int
Val v8 = int_u2(256);          // int_u2
```

Note, to avoid compiler errors, ALL basic types accounted for (especially ints) otherwise, overload ambiguities!!

How does Val handle all types?

- Overload the constructor on all basic types

```
class Val {
    Val (int_1 v): tag('s') { u.s = v; }
    Val (int_2 v): tag('i') { u.i = v; }
    Val (int_4 v): tag('l') { u.l = v; }
    ...
    Val (const string& s);
    Val (const Tab& t);
    ...
    Val (real_4 v): tag('f') { u.f = v;}
    ...
};
```

Getting Values Out

- Just ask for a value, and converts to static type of variable you are using

```
Val v = 17;  // v contains int
int_4 i4 = 0;
i4 = v;        // gets int_4 out!
  // or
int_4 mm = v; // more direct
```

Just get the value you want out of the Val!

More Getting Values Out...

```
Val v = 17;
int_u4 i4 = v;    // convert out to 17ul
real_8 r8 = v;    // Convert out to 17.0
real_4 r4 = v;    // convert out to 17.0f
size_t s  = v;    // convert to size_t(17)
```

Converts the value inside the Val to the static type requested

AS C/C++ would do the conversion without Val in mix

[Principle of Least Surprise]

```
Val vv = 255.8; // a real_8
int ii = vv;    // truncates to 255 as C would!

Val uu = -1;    // an int
int_u1 ll = uu;  // makes into 255 as C would!
```

How Do You Implement Casting?

- C++ has a (rarely) used feature: conversion operators

```
class Val {
    operator int_1();  // someone asks for int_1
    operator int_2();
    operator int_4();
    ...
    operator real_4();
    ....
};
```

Conversion from Val to int_1 causes C++ to call operator int_1 method

Conversion Operators in Detail

- When C++ sees code like:

```
Val v = ...
int_u1 i1 = v;
```

It converts this (automatically) to:

```
Val v = ...
int_u1 i1(v.operator int_u1());
```

Construction and Conversions

- These two features of C++ (overloading constructors and conversion operators) make it easy to manipulate dynamic values in C++!!

```
# Python
v = 17
f = float(v)
v = “hello”
```

```
// C++
Val v = 17;
float f = v;
v = “hello”;
```

Python Dictionaries in C++: Tab

- The Tab is the C++ “Python Dictionary”
  - keys of the table are Val (limited to “hashable” keys)
  - values of the table are Val (unlimited, other dicts)

```
# Python
t = { 'a': 1, 'b': 2.1, 3: 'three' }
print t['a']     # LOOKUP, returns 1
t['new'] = 17    # INSERTION, new key-value
```

```
// C++
Tab t = “{'a':1, 'b': 2.1, 3:'three' }”;
cout << t[“a”];  // LOOKUP, returns 1
t[“new”] = 17;   // INSERTION, new key-value
```

Tab Literals

- When constructing a Tab, use a string to specify the equivalent Python literal

```
Tab t = “{ 'a': 1, 'b': None, 'c':[1,2,3]}”;
```

Small parser for Python literals built-in OpenContainers

Pros: Small footprint, written as C++,
no need to embed Python interpreter

Cons: Not standard parser, nor “full Python evaluation”

C++ OpenContainers has "Simple" Python Dictionary Parser

```
// C++: Read a table from a file
Tab t;
ReadTabFromFile("init.table", t);
```

```
# Python: Read a table from a file
t = eval(file("init.table").read())
```

Both can read a Python Dictionary from a file.

Lookups with Tab

- Lookup returns the type Val&

```
Tab t = "{'a':1, 'b':2}";

Val& vref = t["a"]; // A reference to the Val
vref = 17;         // .. changes both t and vref


Val  copy = t["a"]; // A copy of the Val
copy = 100;        // .. only changes copy
```

Lookups with Tab

- Lookup returns the type Val&

```
Tab t = “{'a':1, 'b':2}”;

Val& vref = t[“a”]; // A reference to the Val
vref = 17;          // .. changes both t and vref


Val  copy = t[“a”]; // A copy of the Val
copy = 100;         // .. only changes copy
```

Like C++, references only valid as long as entity exists

Cascading
Lookups, Changes and Inserts

```
# Python
t = { 'a': {'b': 1.1} }
print t['a']['b']   # LOOKUP, 1.1
t['a']['b'] = 7     # CHANGES 1.1 -> 7
t['a']['new'] = 100 # INSERT 'new':100 into a
```

```
// C++
Tab t = “{ 'a': {'b': 1.1} }”;
cout << t[“a”][“b”]; // LOOKUP, 1.1
t[“a”][“b”] = 7;     // CHANGES 1.1 -> 7
t[“a”][“new”] = 100; // INSERT 'new':100
```

```
//*** The C++ works because t[key] returns Val&
```

Arr is the Python List

```
# Python
a = [1, 2.2, 'three']
print a[1]    # LOOKUP via index: 2.2
a.append(400) # APPPEND
```

```
// C++
Arr a = “[1, 2.2, 'three']”; // Use literal
cout << a[1]; // LOOKUP via index: 2.2
a.append(400) // APPEND
```

Why Val/Tab/Arr?

- Three letters: easy to type
  - since Python doesn't even HAVE to specify type
- VALue, TABles, ARRays
- Val is to remind you that, by default, all things are copied by value (deep-copy!)
  - There are Proxy Values that are ref-counted and behave JUST LIKE Python (Advanced topic, see FAQ)
  - `Val v = new Tab(“{'a': 1}”); Val shared = v;`

C++ Libraries feel like Python!

- Design goals of the PicklingTools
  - Make the Python and C++ interfaces
    - simple (not too hard to use)
    - similar (both Python and C++ look the same)
      - Note both C++ and Python MidasTalker same BY DESIGN!! (as are the MidasServer, MidasListener, MidasListener)
  - Make C++ experience with Python Dictionaries as pleasant as the Python Experience
    - considered BOOST any type, not easy enough to use

MidasTalker in C++ (like Python)

```
#include "midastalker.h"
MidasTalker mt("dl380", 8888, SERIALIZE_P0);

mt.open();  // connect!

// Send request to server
Val request = Tab("{'PING': {'timeout':5.0 }}");
mt.send(request);

// Receive response back, wait up-to 4 seconds
Val response = mt.recv(4.0); // Returns None if
                             // no resp in 4

if (response==None) error_out(); // No response?
```

Documentation

- Website: http://www.picklingtools.com
  - FAQ document
  - User's Guide
  - Paper (history and high-level overview) from *New Application Areas in Open Source Software*
    - "Complex Software Systems in Legacy and Modern Environments: A Case Study of the PicklingTools Library"
    - slides from talk available as well

Demo ...