Complex Software Systems in Legacy and Modern Environments: A Case Study of the PicklingTools Library

Richard T. Saunders
Rincon Research Corporation
Tucson, AZ 85711 USA
rts@rincon.com

Abstract

Most complex software systems are written in many languages and utilize multiple frameworks. The PicklingTools is an open-source collection of libraries (see the website at http://www.picklingtools.com*) allowing multiple systems (both modern and legacy) to communicate. The original purpose of the PicklingTools was to allow users to communicate with a legacy product (written with a monolithic legacy framework called Midas 2k) without needing the legacy framework. Since then, the toolset has evolved to become a standalone framework for building applications in the C++ and Python programming languages. This case-study explores how the PicklingTools has been used to evolve existing legacy applications, exploit existing legacy installations, and author new modern applications.*

1. Introduction

Many modern software systems, of any complexity, are written using multiple frameworks and multiple languages. For example, *Mercurial*[1] is an open-source, distributed version control system written mostly in Python, with key routines written in C for speed (the diff portion). Graphical User Interfaces (GUIs) for Mercurial are abundant and written in many different languages/frameworks: pyGTK, Tcl/Tk, Qt4, MacOS[1].

This paper is a case study of the PicklingTools: an open-source collection of socket and file tools written in C++ and Python. The PicklingTools were written to address the complex nature of modern software's usage of multiple languages. This case study explores how the PicklingTools have been a key component in many operational systems by allowing disparate portions (some pieces written in Python, some pieces written in C++) of these heterogeneous systems to communicate easily. Although the PicklingTools are usable in multiple frameworks, they are framework agnostic and can be used without being tied to any monolithic framework.

For more discussion of considerations and alternatives, see Related Work in section 7.

2. Overview

The PicklingTools allow C++ programs to communicate with Python programs via sockets or files and vice-versa. When data is exchanged between C++ and Python with the PicklingTools, the data is always a Python dictionary (frequently just called a table). See Figure 1.

A Python dictionary is a recursive, dynamic, key-value data structure similiar to a Perl hash, or a C++ `map`. Keys, although they are usually strings, can be any basic type. Values can be any type *including other dictionaries*. The values in the tables are accessed via the keys: for example, if `table` is the dictionary from Figure 1, then `table["menu"]["id"]` would have the value of `17`. Python dictionaries are the centerpiece of the PicklingTools as they are the *currency* of all communications.

The PicklingTools collection simplifies the process of sending tables over sockets (UDP or TCP/IP) as well as loading/saving them to files. Tables can be stored or loaded in many different formats: either textual (human readable like the example above) or binary. When Python stores a table in a binary format, the table is said to be *pickled*, thus the name *PicklingTools*.

There are other closely related exchange formats: Javascript Object Notation (JSON)[2] (which is essentially compatible with Python dictionaries) or XML[3]. Python dictionaries were chosen because of their compatibility with OpalTables (see below).

```
{
 "menu": {
   "id": 17,
   106: "File",
   "popup": {
     "menuitem": [
       {"value": "New",
        "click": "CreateDoc()"},
       {"value": "Open",
        "click": "OpenDoc()"},
       {"value": "Close",
        "click": "CloseDoc()"}
     ]
   }
 }
}
```

Fig. 1. Example Python dictionary: The currency of the PicklingTools

```
{
  menu = {
    id = 17,
    "106" = "File",
    popup = {
      menuitem = {
        "0" = { value="New",
          click="CreateDoc()"},
        "1" = { value="Open",
          click="OpenDoc()"},
        "2" = { value="Close",
          click="CloseDoc()"}
      }
    }
  }
}
```

Fig. 2. Example OpalTable: The currency of Midas 2k

3. History

Rincon Research Corporation (RRC) builds Digital Signal Processing applications (demodulation, cleaning noise from signals, etc). RRC uses two frameworks to build these applications: X-Midas (FORTRAN based) and Midas 2k (C++ based).

Historically, X-Midas was abandoned in 1996 to make way for Midas 2k. RRC moved forward, as did other corporations, writing some very useful applications in the Midas 2k framework: GALACTUS and EARTHBOUND to name a few (names have been changed for company proprietary reasons). GALACTUS was notable as it had (and still has) thousands of installations.

In 2001, the Midas 2k development project was cancelled. The customer directed a return to X-Midas. The importance of legacy was felt. The problem was that both X-Midas and Midas 2k were monolithic frameworks: a user must use the entire environment exclusively.

4. Legacy Systems

When Midas 2k was cancelled, a problem quickly emerged: A large number of very useful applications were already written in that framework. The customer, with no future stake in Midas 2k, required that most of the Midas 2k applications be rewritten in X-Midas.

4.1. GALACTUS

One of the more widely used applications, GALACTUS, was written in Midas 2k. GALACTUS took full advantage of features of M2k (threads, copy-on-write) that were not available in X-Midas: this would make porting problematic. More important was the legacy of testing: GALACTUS was a complex application that had been debugged in the field after thousands of hours of work. GALACTUS was stable, well-tested, fast, and did an important job. Rewriting it would have been an insurmountable problem. GALACTUS was given an exemption on rewrite.

GALACTUS is a server-based application. Clients send requests to GALACTUS over a socket and receive responses back over the same socket. Requests are encoded using a proprietary Midas 2k data structure called an OpalTable. An *OpalTable* is a recursive, dynamic, heterogeneous table, very much like Python dictionaries or Perl hashes. See Figure 2.

The OpalTable format, although developed completely independently of JSON or Python dictionaries, has remarkably similar syntax: `=` replaces the `:` and the keys do not need as much quoting.

4.1.2. Currency of PicklingTools

One of the goals of the PicklingTools is to lower the *entry barrier* of using GALACTUS for modern systems written in a combination of Python and C++. X-Midas, at that time, only supported flat data structures—no recursive tables. For multiple reasons, Python dictionaries

were chosen as the currency of choice over OpalTables and XML.

1) *Support:* The OpalTable is a stovepipe Midas 2k construct—outside of Midas 2k, the OpalTable is essentially unsupported. Python dictionaries and XML have excellent support from their respective communities.
2) *Politics:* OpalTables were untenable because they implied Midas 2k usage (which was discouraged by the customer).
3) *Mind Share:* The Python dictionary is a well-understood standard used by many users. The close correspondance to JSON further enhances Python dictionary mindshare. XML has mind-share, but has other issues (see below). OpalTables simply do not have mindshare.
4) *Entry Barrier:* Using Python dictionaries in Python is very easy, as it is a built-in language feature. XML has a non-trivial learning curve[2] and OpalTable documentation is difficult to find.
5) *Correspondence:* Converting between OpalTables and Python dictionaries is trivial. XML has issues because it does not support the key-value correspondance as directly. XML also has schema, interface, and simplicity issues. For more discussion of choosing JSON vs XML, see [2].

4.1.3. The PicklingTools Solution

The PicklingTools were originally designed to allow non-Midas 2k users to communicate with GALACTUS without needing the Midas 2k framework.

At this time, X-Midas was being evolved to allow Python to be used for scripting code and C++ for low-level code. Given that these were the languages most X-Midas programmers would be using, it was clear that users of GALACTUS would want to be using C++ and Python to send requests.

The center of communications in GALACTUS is a component called the *OpalDaemon*. All client/server socket interactions happen through this component. There are two major problems with the *OpalDaemon* from the perspective of non-Midas 2k users: The first is that the *OpalDaemon* only understands Midas 2k serialization (binary or textual OpalTables). The second is that the socket protocol used by *OpalDaemon* is unintuitive: A bug in the VMWare VMS TCP/IP stack did not allow full duplex sockets—this "feature" keeps X-Midas using something called dual socket mode.

To move forward, the PicklingTools would have to:

1) Augment *OpalDaemon* to understand different formats.
2) Create easy-to-use clients in Python and C++ that encapsulate the unintuitive socket protocol.

The low-risk choice was to allow the *OpalDaemon* to understand Python dictionaries. Python was becoming the "scripting language of choice" for the X-Midas community (as Python replaced the internally developed scripting language) and there was an obvious correspondance between OpalTables and Python dictionaries. Preserving backwards compatibility with the original *OpalDaemon* was still important: there were still many applications written in Midas 2k and these apps still had to communicate with GALACTUS without modifications.

The first version of the PicklingTools initiated the following:

1) GALACTUS replaced the *OpalDaemon* components with the *OpalPythonDaemon*. The first version of the *OpalPythonDaemon* was simply the *OpalDaemon* augmented to understand the Python Pickling Protocol 0 as well as Midas 2k serialization. To remain backwards compatible, adaptive serialization was used to distinguish between the two serializations.
2) C++ and Python users use the MidasTalker client to communicate with GALACTUS.

With these changes, non-Midas 2k users are able to communicate easily with GALACTUS via Python dictionaries. See Figure 3.

4.1.4. Python

Python, out of the box, comes with all the tools necessary to interface to GALACTUS, including:

1) Python dictionaries, a part of the language.
2) The socket library, a standard built-in (import socket).
3) Native Python serialization, called *pickling*, a standard built-in (import cPickle).

The client, called the MidasTalker, is written in raw Python and requires no other extensions. The most time consuming chore was emulating the socket protocol that the server uses, but even that was straight-forward.

An important aspect of the MidasTalker is that it is written in *raw Python*—there are no dependencies on any non-standard external libraries, in C or otherwise. This means a user can write a Python client to talk to GALACTUS without needing any special frameworks; communicating with GALACTUS is as simple as an import, an open, and a send. See Figure 4.

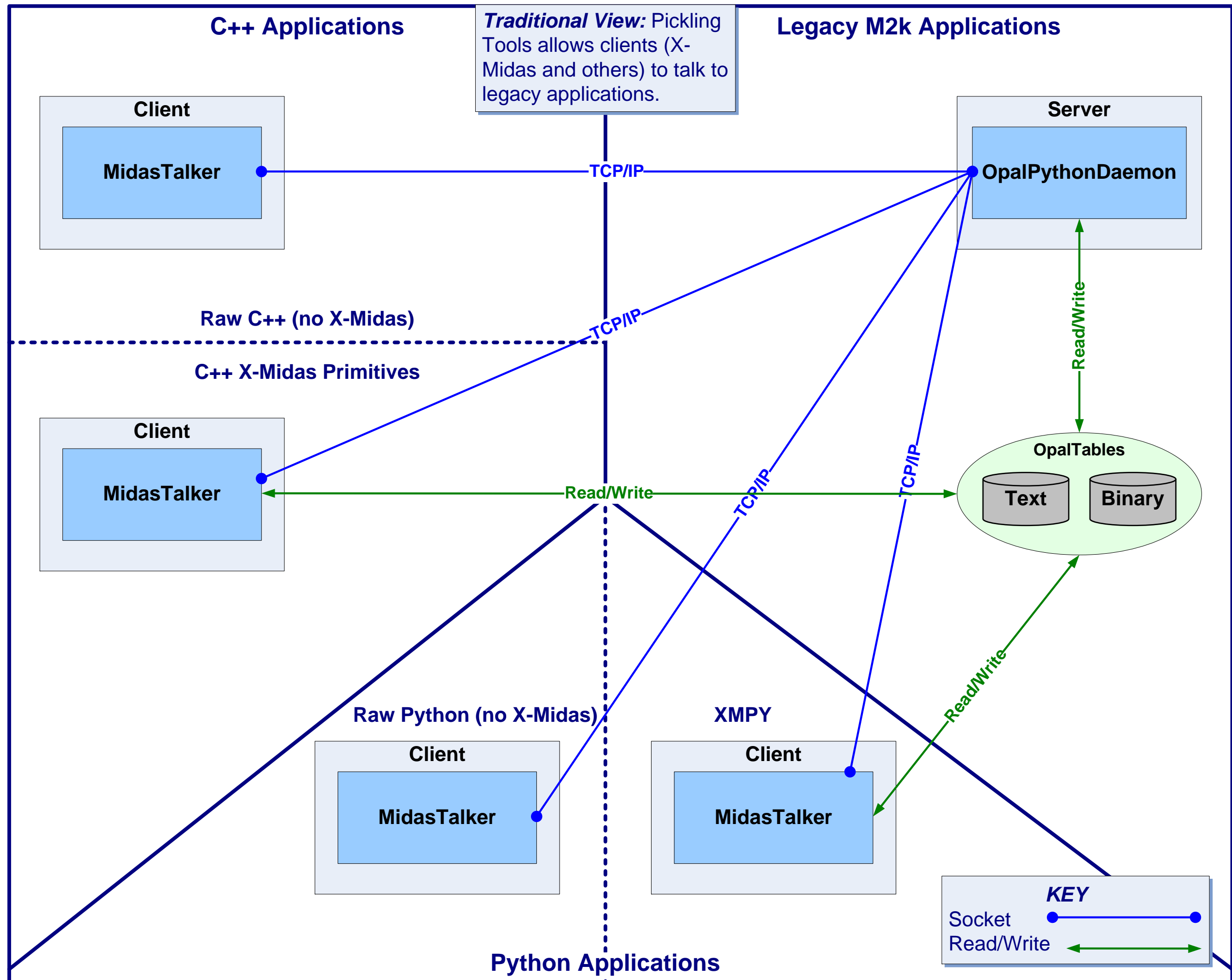

Fig. 3. Traditional View of PicklingTools

```
from midastalker import *
mt = MidasTalker("host", 8888, SERIALIZE_P0, DUAL_SOCKET)
mt.open()

request = {'REQUEST': {'ping': 100.1} }
mt.send( request )
response = mt.recv(5.0)   # 5 second timeout
if response['STATUS'] == 'OKAY' : done()
```

Fig. 4. Example usage of a MidasTalker in Python

4.1.5. C++

C++, out of the box, needs a few more tools to interface with GALACTUS:

1) Python dictionaries are emulated through an open-source library called OpenContainers[4], included with the PicklingTools distribution.
2) The UNIX socket libraries come standard on most distributions.
3) The Python Pickling modules had to be written from scratch to emulate Python serialization. They are included in the PicklingTools distro.

```
#include "midastalker.h"
MidasTalker mt("host", 8888, SERIALIZE_P0, DUAL_SOCKET);
mt.open();

Val request = Tab("{'REQUEST': {'ping': 100.1} }");
mt.send(request);
Val response = mt.recv(5.0);  // 5 second timeout
if (response["STATUS"]=="OKAY") done();
```

Fig. 5. Example usage of a MidasTalker from C++

The C++ libraries are more cumbersome than their counterpart. It's difficult to emulate Python dictionaries, a dynamic construct, in a statically typed language like C++, but the OpenContainers libraries allow C++ users to manipulate data structures that are similar to Python dictionaries. Emulating the same semantics as the Python Pickling Protocol 0 required reverse engineering the Python serialization format. Although the Boost libraries[5] were available at this time, it wasn't clear they contained the pickling code. See Related Work for more discussion.

A minor goal of the PicklingTools is to make the C++ experience similar to the Python experience. The C++ MidasTalker looks and feels very much like its Python counterpart: See Figure 5.

The original goals of the PicklingTools were satisfied. Any user could interface to GALACTUS easily from Python or C++, without requiring X-Midas or Midas 2k. By having a toolset that wasn't bound to any particular framework, users could easily move between them.

4.1.6. Python Extension Modules in C and Embedding Python

For Python, a common solution to the "How to connect Python and C/C++" problem is to wrap the C/C++ in a C extension module (i.e., wrap the C code with some special code to be callable from Python). Python can then easily call the needed functionality[6]. Another potential solution is to embed a Python interepreter within a C program[6] so that the C/C++ program can call arbitrary Python code.

Why didn't the PicklingTools consider these solutions?

1) *Linkage Issues Complicate Usage:* C extension modules for Python imply the user has to worry about linking C code to their Python interpreter. Linking, esp. with shared objects, is full of pitfalls in a diverse environment: different OSes (Red Hat 9.0, Enterprise 3, 4 and 5), different compilers (GNU gcc 3.x, 4.x, Intel), different models (32-bit and 64-bit), different versions of Python on the same machine (2.2, 2.3, etc). Linking issues plague installs when sharing (disks, code) tends to cross machine-boundaries.

The client needs to be *as simple as possible* to avoid linkage issues: All PicklingTools Python code is *raw Python* to avoid any linkage issues. Even most of the C++ PicklingTools code, especially OpenContainers, is *inline* code so the user can just `#include` the appropriate modules without having to link it (C++ and related tools supports inline code in `.h` files well). Using this model, the user defaults to the link environment of the surrounding application.

A major concern of the PicklingTools is to reduce the entry barrier for modern applications: worrying about linkage issues raises the entry barrier to talk to legacy applications.

2) *Libraries Are Low Risk Compared to Extending/Embedding Python:* Inserting a few libraries into the *OpalPythonDaemon* is low risk: they have a small memory footprint and changes are isolated to one component.

Converting all of GALACTUS to a C extension module to be called by Python is problematic because it is such a large task: the size of such an enormous undertaking is inherently risky.

Embedding a Python interpreter is worrisome due to of linkage issues. Most legacy Midas 2k applications use many threads (50-200): these threads tend to be address-space hogs. The address space used by a Python interpreter is non-trival and can "bloat" a legacy application. GALACTUS is especially sensitive to address-size bloat as it stresses the limits of a 32-bit install.

To mitigate risk and keep the the entry barrier low for the legacy applications under consideration, Python extensions or embeddings do not make sense.

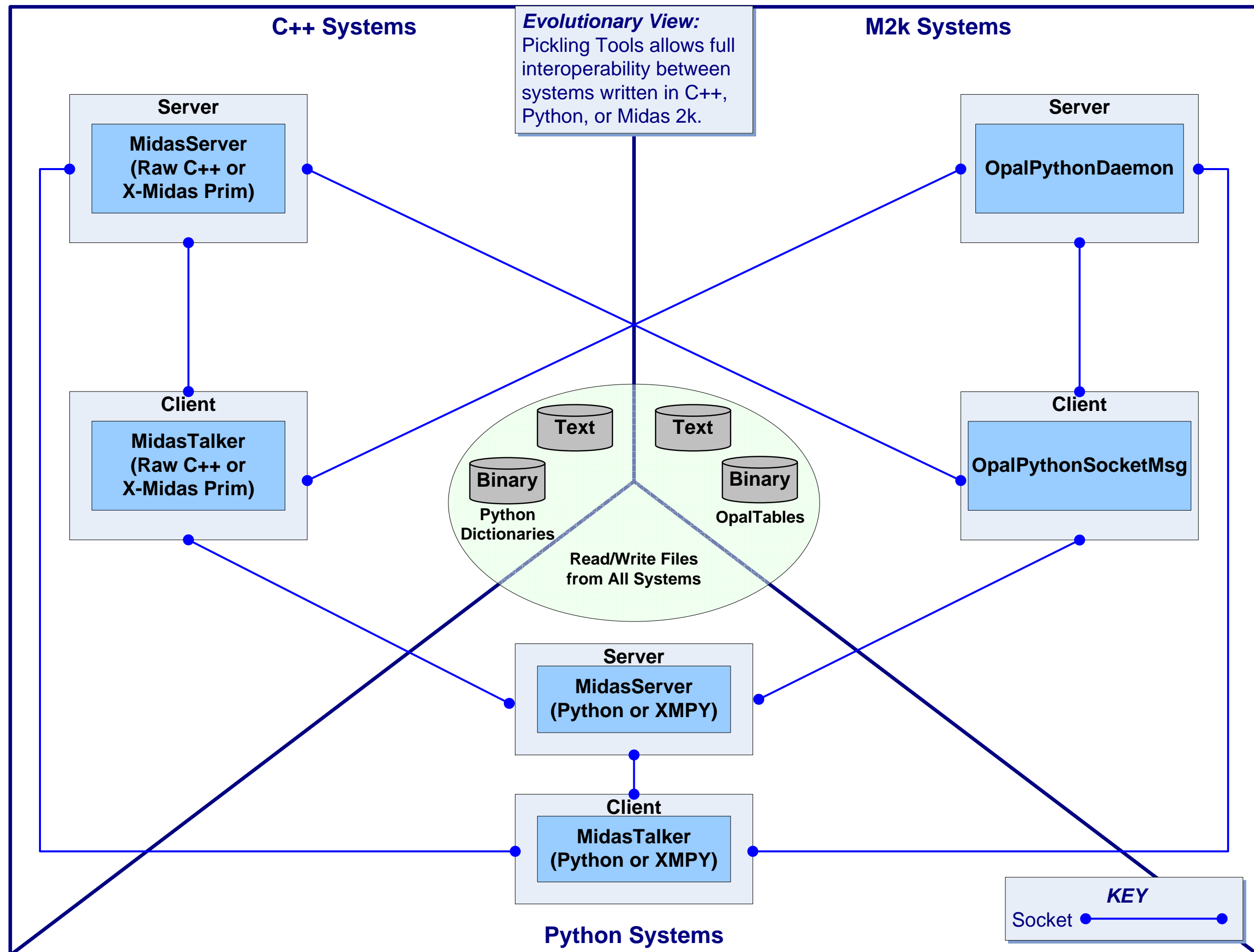

Fig. 6. Evolutionary View of PicklingTools

5. Evolving Systems

GABRIEL was a new X-Midas application tasked to replace an existing Midas 2k application. GALACTUS was just one of the systems that GABRIEL needed to communicate with. The developers of GABRIEL decided to use the C++ portion of the PicklingTools to talk to GALACTUS, but enjoyed the experience so much that they adopted the toolset for the entire application. The GABRIEL developers used Python dictionaries as the currency of their system, sending them through X-Midas pipes or writing them to files on disk. This application heralded the way for other X-Midas applications to use the PicklingTools.

Each new system using the PicklingTools evolved the library, incrementally adding new features.

1) GABRIEL must communicate with a Midas 2k system that still delivers textual OpalTables in files. The PicklingTools was expanded so that it could handle textual OpalTables from Python and C++. In other words, a new serialization was added.
2) SSURF was a X-Midas follow-on to the GABRIEL system. SSURF was a more complex system using PyQt GUI, GALACTUS, and an assortment of XMPY (Python) and C++ pieces. It was very clear that a system using lots of MidasTalkers also needed MidasServers. Thus, SSURF pushed the development of the MidasServer in Python and C++. See Figure 6.
3) TERRAX is an X-Midas system that must communicate with a EARTHBOUND, a legacy system written in Midas 2k. EARTHBOUND cannot be replaced, modified, or changed in any way. Simply speaking, it is untouchable because of money and geography issues: it works and it cannot afford to be rewritten. TERRAX, however, must communicate with EARTHBOUND to get

Fig. 7. A setup of Arkham Horror: A complex table-top game with many components

the information it needs. To move TERRAX forward, the PicklingTools was augmented to handle:

a) UDP
b) M2k binary serialization

The PicklingTools framework delivers UDP OpalTables from EARTHBOUND to X-Midas as Python dictionaries so TERRAX can process the information. Both C++ and Python can handle UDP, although only C++ can handle M2k serialization.

4) NOVA is an X-Midas system that talks to GALACTUS. Unlike other systems, NOVA is an incredibly high-volume system that stresses GALACTUS to the limits. The default serialization between systems is Python Pickling Protocol 0: it is a protocol compatible with just about all Python interpreters, but it is a slow ASCII serialization protocol. To get NOVA the performance it needed, the Python Pickling Protocol 2 (which comes standard with most modern Python interpreters) was implemented: it handles binary data faster with less overhead: 10x–100x faster. In the end, NOVA used the slightly faster M2k binary serialization (due to compatibility issues), but it pushed the PicklingTools forward to be able to handle Python Pickling Protocol 2.

The PicklingTools has quickly become very important in allowing new systems to be written in X-Midas (using newer technologies in Python), while still leveraging the legacy software.

6. New Applications

Although historically both X-Midas and Midas 2k have been involved in every system using the PicklingTools library, neither is required: all the tools can be used from raw Python or raw C++. In this way, complex applications made up of C++ and Python can be easily built using Python dictionaries as the currency of the system. See Figure 6: it gives the Evolutionary view of the world.

6.1. Arkham Horror

In Spring 2008, the Software Engineering class from the Department of Computer Science at the University of Arizona used the PicklingTools to build a complex game.

The game Arkham Horror is a complex table-top board game. It has numerous complicated rules for play and is time-consuming to set-up and take down. See Figure 7. It also requires that everyone be at the table to play. If the computer manages the board and the

Fig. 8. Sample Monster Card from Arkham Horror

```
{
    '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'
}
```

Fig. 9. Sample (Simplified) Encoding of Monster Card from Figure 8

rules, the players are free to enjoy the game. Using networking allows distant players the option to play. Computerization makes the game more accessible.

The goal of the Software Engineering class was to implement, as an entire class project, a computerized version of Arkham Horror.

6.2. Game Architecture

The game consists of hundreds of cards, each of which changes the rules slightly or asks the player to face some minor challenge. There are also, at the time of this writing, at least six expansion sets which add new cards to the game. One architectural goal was to make it easy to add the new cards to the game without having to modify client or server code (see below).

The basic idea was to make a "Prolog-like" rules system where all the cards of the games are encoded as either XML or Python dictionaries, and the game engine would process those, changing the rules dynamically as the cards were drawn. For example, a sample monster card for the game looks like Figure 8 and might be encoded (simplified version) as in Figure 9.

The main reason the PicklingTools were used was

the desire for transparency in the architecture: all cards had to be easily human-readable without needing excessive tools. The developers would be continually updating, evolving, reading, and manipulating hundreds of cards. If the cards were stored as textual Python dictionaries, then all forms of manipulation would be easy: human reading/writing as well as Python/C++ manipulation. One back-end alternative was XML, but XML is not particularly readable without special tools[2][3]. XML also requires some external tools to use from C++, Java, or Python[2][3][7]. Manipulating Python dictionaries from Python is easy because they are built-in to the language (the OpenContainers give a similar experience, by design, in C++: See Related Work).

The architecture for Arkham Horror was a Model-View-Controller[8]. The main game engine, which encodes the state of the game (where players are, the health of each player, etc.), is implemented as a PicklingTools server in C++ (as a MidasServer). Each player playing the game is implemented as a client (a MidasTalker) that talks to the server to obtain and display the current state of the game. The original clients were text-based, with a Java and/or Web GUI to be put on top.

The PicklingTools library was used because it allowed:

1) Easy socket communications [players talking to game engine].
2) Easy data representation [Python dictionaries].
3) Easy manipulation of data [textual dictionaries read from files].
4) Easy choice of languages [C++ or Python].

The Python dictionaries back-end made PicklingTools viable and the simple-to-use socket manipulation made the PicklingTools a clear choice.

During initial development, the goal was to use Python as a prototyping language to flesh out the communications between the the clients and the servers. The final goal was to implement the engine in C++ so it could have multiple concurrent threads, with a thread for each player (Python has issues with true concurrent threads because of the Global Interpreter Lock[7]).

Due to a miscalculation by the instructor, the skill-level of the class with C++ and Python was much less than expected. Nevertheless, the class was able to pick up the PicklingTools and make progress. However, since the class was primarily Java oriented and one group was able to quickly develop an initial version of the PicklingTools for Java, the class switched the engine and graphical clients to Java. This demonstrated that there is a need for the PicklingTools to further evolve by adding support for Java. There are plans to cleanup and fully incorporate the Java support in a future release. The initial work in Python and C++ helped lay the foundation of the game, and the class continued using the Java PicklingTools and Python dictionaries as the currency for the game.

A full version of the game was not expected since it was a single semester class. With this time constraint in place, a rough version was developed.

There were overtures to Fantasy Flight Games (the owner of the copyright for Arkham Horror) to cooperate and produce a real product. Unfortunately, the version developed by the class was not far enough along for them to be interested. Fantasy Flight maintains their copyright (to the point of telling others to take materials off of websites) so it's improper to distribute the version written by the class.

7. Related Work

A popular solution for mixing C++ and Python are the Qt4[9][10] and PyQt[11][12] frameworks. Qt4 is a very complete and mature C++ framework with an emphasis on GUIs, but it also offers support for networking and serialization. XML is pervasive in the Qt framework and tends to be the serialization of choice: For example, the GUIs from Qt Designer are stored in files as XML. Qt also offers a homegrown binary serialization through `QDataStreams`[10]. The Python framework PyQt is produced directly from the Qt C++ using SIP[13], a tool for automatically generating Python bindings for C and C++ libraries. The integration between PyQt and Qt is tight, as the SIP produces simple wrappers to call the C++ from Python[14]. SIP produces those wrappers in an automated way.

Qt is a viable solution, but licensing may play a role. Certain editions of Qt are available under a commercial license, but there is also an Open Source version. Until recently (2006-2007), there were questions as to whether the PyQt bindings were under GNU General License (GPL) version 3 or GNU Lesser General Public License (LGPL) (version 2.1). If you are willing to buy the commercial version or are not worried about the GPL, Qt might be a viable solution.

From a PicklingTools' perspective, Qt was never really an option. Qt was available, but the licensing issues seemed problematic from a corporate level. More importantly, Qt has historically been more of a GUI toolkit, and it wasn't clear when RRC was shopping around (2001) that Qt was the right tool: It tends to be a framework that you embrace completely rather than

piecemeal.

Another popular solution for mixing C++ and Python are the Boost C++ libraries[5][15]. The Boost libraries offer heterogenous containers (using the `any` type[5]) like OpenContainers (using the `Val` type[4])—this allows the C++ user to use "python-like" data structures. The `Boost.Python` libraries also offer serialization libraries that are compatible with Python Pickling. The Boost libraries are very general, but slightly harder to use. The OpenContainers and PicklingTools chooses the opposite tact: the design goal is ease of use, at the price of generality. For example, the Boost libraries can work with any C++ classes, but require augmenting C++ classes with code and annotations to support Pickling[15]. The PicklingTools, on the other hand, give you a fixed set of datatypes (all ints, floats, complexes, strings, None, tables, lists) but make those easy to use and pickle[16].

A subgoal of the PicklingTools is to make the C++ experience with heterogeneous containers as pleasant as the Python experience. The Boost containers are less "Python-esque" than the OpenContainers and therefore less desirable. The Boost Python libraries may have been a reasonable option at the time, but they seemed clumsy to use.

Another solution to mix C++ and Python easily is to use tools like SIP[13] and SWIG[7] to wrap C/C++ code automatically, rather than write these bindings by hand using [6]. This requires linking C/C++ code into the Python interpreter, which (as discussed in section 4.1.6) can be problematic if you aren't mindful of linking issues.

A critical point to note was that the PicklingTools evolved slowly over time (8 years). Each feature added value incrementally and without much risk. It's important to understand that this one consideration explains why much of the Related Work never made much sense to explore extensively; embracing Qt/PyQt, Boost, or SWIG at any step would have been a higher risk proposition. When developing applications under severe time pressures, the "small change/low risk" option is very appealing.

8. Conclusion

The PicklingTools started life as a toolset to ease the transition from a legacy system to more mainstream toolsets. As it evolved, it grew into a full-fledged standalone toolkit allowing users to write applications from scratch.

The PicklingTools have been used to build and augment real systems (GALACTUS, GABRIEL, SSURF, TERRAX, NOVA) that run continuously and have severe time and memory constraints. By way of example: SSURF is 378000 lines of Python/C++ code and uses 400+ quad-core machines. NOVA is 406000 lines of Python/C++ code and uses 120+ quad-core machines.

The PicklingTools offers two major features over other toolsets. First off, much of the value of the PicklingTools has been supporting multiple serializations (many flavors of binary and text) so that disparate systems could talk. New systems were able to come online because the PicklingTools were able to bridge the gap. The other major feature is the ease of usage: The PicklingTools went to great lengths to provide simple and consistent interfaces between C++ and Python. The ease of manipulation of Python dictionaries from both Python and C++ has been critical to its adoption by real-world projects.

The toolset continues to evolve and is frequently updated (see the web site).

Thanks to Rincon Research Corporation for allowing the PicklingTools to be open-sourced, and thanks to the University of Arizona for allowing usage in the Software Engineering class.

References

[1] B. O'Sullivan, *Mercurial: The Definitive Guide*. California: O'Reilly Media, Inc, 2009.
[2] Miscellaneous, *JavaScript Object Notaton*. http://json.org, 2004.
[3] T. Bray, J. Paoli, C. M. Sperberg-McQueen, E. Maler, and F. Yergeau, "Extensible markup language (xml) 1.0 (fourth edition)-origin and goals," *World Wide Web Consortium. http://www.w3.org/TR/2006/REC-xml-20060816/sec-origin-goals. Retrieved on October 29 2006*, 2006.
[4] R. T. Saunders, *OpenContainers: A Portable, Thread-Neutral Library*. www.amalgama.us/oc.html, 2004.
[5] B. Karlsson, *Beyond the C++ Standard Library: An Introduction to Boost*. California: Addison-Wesley Professional, 2005.
[6] G. van Rossum and F. L. Drake, *Extending and Embedding the Python Interpreter, Release 2.3.4*. python.org, May 2004.
[7] M. Lutz, *Programming Python, Third Edition*. California: O'Reilly Media, Inc, 2006.
[8] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns*. California: Addison Wesley, 1994.
[9] M. S. Jasmine Blanchette, *C++ GUI Programming with Qt 4 (2nd Edition)*. Boston: Prentice Hall, 2008.
[10] T. (Nokia), "Qt online reference documentation," 2009.
[11] M. Summerfield, *Rapid GUI Programming with Python and Qt: The Definitive Guide to PyQt Programming*. California: Prentice Hall, 2008.
[12] R. Computing, *PyQt Whitepaper*. riverbankcomputing.co.uk/software/pyqt/whitepaper, 2009.
[13] ——, *SIP Reference Guide*. riverbankcomputing.co.uk/software/sip/intro, 2009.
[14] ——, *PyQt v4 - Python Bindings for Qt v4*. riverbankcomputing.co.uk/static/Docs/PyWt4/pyqt4ref.html, 2009.
[15] D. Abrahams and R. W. Grosse-Kuntsleeve, *Building Hybrid Systems with Boost.Python*. Dr. Dobbs Journal, July, 2003.
[16] R. T. Saunders, *The Pickling Tools User Guide*. http://www.picklingtools.com/UsersGuide.txt, 2004-2009.