Mobile Computing with Python

James “Wez” Weatherall & David Scott,
Laboratory for Communications Engineering,
Cambridge,
England,
{jnw22, djs55}@eng.cam.ac.uk

Abstract

This paper describes MoPy, a port of the Python 1.5.2 interpreter to the Psion 5/Epoc32 platform and Koala, a CORBA-style Object Request and Event Broker implemented natively in Python. While primarily a direct POSIX-based port of the standard Python interpreter, MoPy also adds thread, socket and serial support for the Psion platform. The Koala ORB is designed particularly with MoPy in mind and aims to support interoperation of devices over the low-power Prototype Embedded Network. The low-power requirements of PEN impose severe bandwidth and latency penalties to which conventional RPC technologies are not typically well suited.

1 Introduction

Recent advances in wireless communications and embedded processor technology have paved the way for tetherless, intelligent embedded devices to enter into everyday use. Already technologies such as BlueTooth[2] and WAP[19] enable users to read their email, browse the web and link their various computational devices without physical network connections. This paper describes some key features of Koala, an Object Request and Event Broker written natively in Python and designed for use on handheld and wireless devices, and MoPy (Mobile Python), a Python interpreter for the Psion 5 Series[14].

The PEN project[1] at AT&T Laboratories Cambridge explores the future possibilities of embedded low-power wireless communications support in both conventional computer equipment and everyday appliances, with a view to providing totally ubiquitous interaction between devices. The prototype units use the 418MHz band and provide 24kbps throughput over a 5 metre radius, with power management allowing a unit to operate for several years on a single cell, subject to environmental conditions. The PicoRadio[15] and Hyphos[13] projects are examples of systems with similar design goals to PEN.

One example of ubiquitous device interaction is that of a generic remote controller interacting with its environment. The controller would act as a mobile, untethered user interface, with wireless access via PEN to all the locally available devices. These might

range from simple light and temperature sensors, through lamps and window blinds, to a PC's MP3 jukebox or a hi-fi.

An ideal platform to act as the remote controller is the Psion organiser. The device is compact, low-power, and already provides a number of useful applications. A PEN node may be attached to the serial port of the Psion and used as a network interconnect. The Psion then runs client-side software which can use the node to scan the local area for other nodes and to communicate with them.

The Psion can use the connection to pull email from its user's folders when he is in the office, and can then push that email out to a display in his car, all while tucked away in his briefcase or sitting on his desk. It can use the contextual information inherent from the short range of the radio link to decide when the user is at home, on their bike or in the office for example, and reconfigure neighbouring equipment accordingly.

1.1 Follow-me-MP3

A popular demonstration for many distributed multimedia and sensor systems is that of "follow-me audio". In this example, a user roams around a building populated with speakers connected to networked PCs. A central MP3 archive feeds audio to the speakers nearest the user, across the network, so that their personal audio preference accompanies them wherever they go.

Using MoPy and Koala, we can extend this example. Our user carries his standard Psion 5 organiser, as he would normally, except that his organiser has a tiny PEN node attached to it's serial port[1]. An example screen-shot is given in figure 1.

As the user moves around the building, the PEN node will be in direct contact with only a few devices — only those within a 5-10 metre radius — allowing it to establish a rough idea of proximity. The Psion can monitor this proximity metric and contact the nearest speakers, telling them to connect to the MP3 jukebox. The previously used set of speakers can disconnect automatically when the Psion is out of radio range, or can be made to disconnect by an RPC to the speaker, through a gateway into the wired network.

Because the Psion is independent of the wired infrastructure, the user can have a similar but separate network within his home. The mobile device can detect that it is in the home and reconfigure accordingly, to use the home jukebox rather than the (perhaps unreachable from home) office one, for example.

1.1.1 Follow-me-MP3 Implementation

The Follow-me-MP3 example has been implemented as a text-based application running under MoPy on the Psion 5mx and under the standard Python distribution for Linux. The devices communicate via PEN nodes attached to their serial ports. The Linux device runs an MP3 jukebox and a "speaker" application, both made available to the Psion over the PEN network using the Koala ORB described below. The Psion then runs a controller application which imports interfaces to the jukebox and speaker and accesses the two entities via the Koala ORB.

[1]The current PEN prototypes are almost the same size as the Psion itself. In principle, though, there is no reason why very tiny embedded versions cannot be built

Figure 1: Screen-shot of a Wireless Follow-me-MP3 Controller

In this system, our primary interest is in the design of the Koala ORB, which interfaces between a high-level communications model and the low-level PEN network interface. This is supported on the Psion platform by the development of the MoPy interpreter, which provides a standard python environment, avoiding the need for a separate development environment and/or Koala implementation.

1.2 Further Applications

In addition to the roaming MP3 example described above, several other applications have been prototyped using the system, of which two are described below.

1.2.1 Home Control

A rudimentary home control system has been built using the PEN, MoPy and Koala systems combined. For this application, home appliances such as lights, blinds, sensors and switches are connected directly to PEN nodes. Each device uses the standard Koala protocol to make itself available via the PEN network.

When a Psion running MoPy and Koala is introduced into this environment, the PEN-enabled appliances may then be monitored and/or controlled by the mobile Psion device. In particular, because of the ease with which new fragments of Python code

may be introduced into the system at run-time, it is straightforward to add automated rules to control home appliances according to personal preference. This capability is currently the subject of further research.

1.2.2 Slaved Remote Controller

A simple "thin client" mobile remote controller has been built, similar in operation to that used in the ParcTAB[18] project. The device is a dedicated user interface with a black-and-white LCD display and three buttons attached to a PEN node, through which it makes its features available. The controller device is then "slaved" to a nearby PC, which runs the software on behalf of the controller and manipulates it using Koala and PEN.

This system works well in an office environment, where PEN base-stations can be placed so as to provide complete coverage of the environment, so that remote controllers operate seamlessly throughout. Because PEN is an ad hoc network, thus requiring no infrastructural support or pre-configuration to operate, the Koala objects provided by these remote controllers may also be accessed by Koala programs running on mobile devices such as the Psion. This allows the remote controllers to be used in a wider variety of ad hoc environments.

2 Koala

The Koala ORB is an Object Request and Event Broker designed to support universal compatibility between devices in an efficient manner. It is implemented partially as a minimal server library written in C for the PEN platform[2], and as a more complete client and server library written in Python.

The ORB is deliberately designed to be functionally equivalent to the CORBA[11] ORB standard but, because of its simpler design and the choice of implementation language, it is a great deal easier to experiment with. Because Python is cross-platform, the bulk of Koala can be built on a PC and the resulting code files transferred to the Psion to be run.

For most of the applications we are building, Psion or PC devices act as the user interface, hosting Koala stub classes for different types of device and importing object references from nearby PEN nodes. The PEN devices run servant objects which can be manipulated by the PC or Psion remotely. For example, in the roaming audio example given in Section 1, our user can cause their audio to be Played, Paused or Stopped, as with conventional audio equipment. The steps involved, using Koala running under MoPy, might look like:

1. `orb = Koala.ORB()`

   The program creates a Koala ORB instance through which to contact remote Koala objects.

[2]Use of a Python interpreter running under PEN was considered, but PEN's threading model and limited memory would have required a Stackless Python[17][16] port, and would have been a prohibitive development effort

2. `orb.add_transport(TCP_Transport)`
   `orb.add_transport(PEN_Transport)`

   The program registers some previously-imported "transport" modules with the ORB. These modules provide a standard API to a variety of underlying networks, through which the ORB may use them.

3. `PEN_trader = PEN_Trader.Trader(orb)`

   The program on the Psion creates a new PEN Trader. This trader monitors the local radio channel via the PEN node connected to the machine's serial port and populates the trader with all the Koala objects accessible remotely via PEN.

4. `jukebox = PEN_trader.FindOne("AudioSource", [])`

   The Psion imports an `AudioSource` object from the local PEN trader. This trader contains object references for all devices accessible locally via PEN.

5. `jukebox = jukebox._narrow("AudioSource")`

   The imported reference is of type `Object`, so it must be `_narrow()`ed to `AudioSource` before it can be used.

6. `jukebox.Play()`

   The jukebox is made to start playing audio using the `Play()` method. The jukebox might also support `Pause()`,`Stop()`, `FastForward()`, etc methods.

2.1 Object-Oriented RPC

As has been mentioned previously, the Koala ORB borrows heavily in its design from the OMG Common Object Request Broker Architecture. CORBA-based systems consist of interfaces specified in standard OMG Interface Definition Language[8], and client and server side stubs to perform marshalling and unmarshalling of Remote Procedure Calls.

Several CORBA-compliant ORBs are available with "mappings" (implementations) for the Python language. Two popular implementations are Fnorb[7] and omniORBpy[12]. omniORBpy uses the same C++ back-end as the original omniORB C++ mapping for performance reasons. Conversely the Fnorb ORB is implemented, with the exception of some low-level marshalling routines, entirely in Python. In our experiments the performance of the system as a whole is dominated by the latency and bandwidth limitations of the PEN network rather than by the ORB itself. Fnorb woudl therefore have been the preferred ORB on which to base our experiments. However, we also require that the ORB should operate on a resource-limited device such as the Psion. For this reason we wished to avoid using a full CORBA ORB implementation, and instead chose to build a simple CORBA-like ORB from scratch.

In Koala, interfaces are currently specified in scripts via a well-defined data structure, rather than from IDL. Interfaces are registered with the ORB at run-time, both at the client and server sides. It will automatically generate appropriate stub code on demand, so that the code only exists when it is needed, saving space. Koala doesn't currently have an IDL compiler, since it is sufficiently straightforward to generate the

Python data structures by hand. It was decided that the effort required to build or modify an IDL compiler was not warranted.

The interface definition structure is also used when dealing with incoming method invocations, to establish how to unmarshal the parameters. Python, being a scripting language, makes this process fairly trivial, since an incoming RPC can be quickly unmarshalled and dispatched using a single `apply()` call.

Method invocations are marshalled via a standard marshalling engine which supports most of the standard OMG IDL types, plus a few new ones. The marshaller is designed to produce compact output as speedily as possible and achieves respectable results.

2.2 Events

One of the experiments we have performed with Koala is to add native event support to the ORB. Any Koala object can be a source of events — asynchronous notifications of a change in their state. Clients can then "watch" the object for changes in its state. The ORB manages this operation as efficiently as possible and notifies the client via callback when the state next changes.

Events may also contain values (current at the time of notification) for the object's attributes. If an interface is derived from the special "Watchable" interface[6] then its attributes will be marshalled into all event notifications that object produces. This serves two purposes — firstly, that the client need not contact the object after each notification, since it will already have the relevant attribute state information, and secondly that the attribute information received can be synchronised at the server side to ensure consistency.

Several Event Service implementations exist which use RPC as their underlying transport mechanism, allowing them to run over a variety of ORB implementations. These services support events with arbitrary content, either pushed from source to sink or pulled by sink from source. There are two main problems with this approach:

Firstly, the use of events with arbitrary content makes it impossible to ascertain in general where the event originated or what caused it. This in turn can make it difficult to reason about the interactions between events. Koala attempts to limit the scope for confusion by stipulating that events always indicate a change in the state of zero or more attributes of a Koala object.

Secondly, in a high-latency environment such as a local-area PEN network or a wide-area network such as the Internet, a separate thread must be created by the event source for each registered event sink, to ensure that events are notified to sinks in parallel. Serial notification would result in too significant delays and hence performance degradation if used with a high-latency network. Koala instead provides a high-level "watch" interface, under which the ORB may then use the most appropriate event transport for the underlying network. This allows events to be transferred by standard RPC for example, or, as is the case with several of the PEN devices we have prototyped, by passing the attribute state information in periodic broadcast messages.

An example device using the Koala event mechanism is a wireless mouse (see Figure 2). The mouse supports three attributes through which its position and the states of its buttons may be obtained. The mouse may either be a primitive device which

```
interface Mouse : Watchable
    attribute unsigned long Xpos;
    attribute unsigned long Ypos;
    attribute unsigned long ButtonState;
};
```

Figure 2: An example interface to a wireless Mouse

simply transmits a broadcast datagram via PEN whenever an attribute changes, or may be a more complex device which accepts incoming `wait()` RPC calls and blocks them until its position changes. Which of these techniques is used depends on the environment for which the mouse is designed. Programs using the mouse need not be aware of the distinction in order to use the device, because of the attribute-watching mechanism provided by Koala.

2.3 Asynchronous RPC

The PEN radio layer is a high-latency transport, partly as a consequence of its low-power operation. As a result, sequentially dispatching multiple method invocations to a remote object can be prohibitively time-consuming, since each request is only dispatched when the previous request has been received.

Very often, such invocations could actually have been dispatched in parallel, in which case an obvious solution is to spawn a separate thread for each request and to join with those threads at some later time. This is a poor approach for two reasons; firstly, spawning a thread can be a very expensive operation, in particular on platforms such as Linux or Psion, on which thread switching is expensive in terms of CPU-cycles (Conventional control-flow processors are not well suited to thread-switching, in fact). Secondly, the return values of each call must be manually saved by each thread and then retrieved once the threads have been explicitely synchronised.

Koala therefore supports asynchronous RPC, a technique designed to be both efficient and simple to use. The calling thread first obtains an asynchronous stub for the object from the ORB. This stub dispatches all invocations made through it to the same socket connection and records the sequence number of the operation, without waiting for the operation to return. Figure 3 shows the difference in timings between synchronous invocation and asynchronous invocation of three independent RPC calls across a moderate-latency network. Figure 4 shows sample Python code to read the status of a Mouse device using asynchronous RPC.

As the ratio of CPU cycles required per invocation to network latency increases, the two cases converge because the time spent by the servant in processing incoming requests is dominant. As the ratio of CPU cycles per invocation to latency decreases, the two cases diverge because the synchronous case involves more round trip delays and is therefore more affected by latency.

The latency per message over a TCP-based network is comparable in maginitude to the processing time required for a typical null-Echo method invocation. As a result, the extra overhead incurred in performing asynchronous invocations reduces performance

greatly. When PEN is used as the underlying network however, latency can exceed CPU time per invocation by several orders of magnitude. The overhead in using asynchronous invocation becomes insignificant in this case.

In general synchronous invocation provides better performance over low latency networks while asynchronous invocation is preferable over high latency networks.

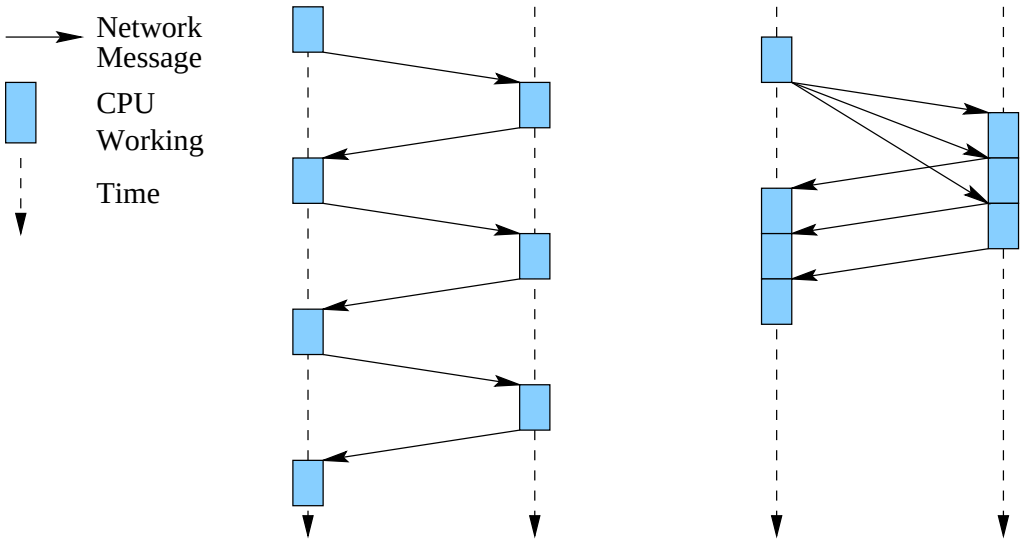

Figure 3: Synchronous versus Asynchronous RPC for three independent invocations

```
amouse = mouse._async_stub()
xpos = amouse._g_Xpos()
ypos = amouse._g_Ypos()
buttons = amouse._g_ButtonState()
...
actual_xpos = xpos()
actual_ypos = ypos()
...
```

Figure 4: Example code for retrieving the initial position and button state of a Mouse device using asynchronous RPC

The client calls methods on the stub as it would the synchronous version, except that all methods now return a callable Python object instead of their normal results. The returned object contains a link to the asynchronous stub and the sequence number of the method invocation. When it is "called" with no parameters, it checks the asynchronous stub to see whether it has already received a response for the operation. If the results of the invocation have not been received then the caller is blocked until they are. If the

result object is deleted without being called, then the any results from that invocation are lost and no synchronisation occurs. Asynchronous result objects also support a `_ready()` method, through which they can be polled for completion if required.

In this way, asynchronous RPC makes it extremely simple to dispatch concurrent requests and to only synchronise with those requests when their results are required, by "calling" the result object. Python's callable objects and lack of strong typing made this a very straightforward feature to implement.

3 MoPy

When the Koala project started, two ports of Python to the Psion existed. The first, by Duncan Booth, was a fairly direct port of Python 1.5.1[3] via the Posix compliance API of Psion, with several useful extensions including a readline-alike library and support for the Psion's touchscreen. The second port, by Otfried Chong[5], was based on the first but extended to have a standard Psion user interface, with the familiar button layout, menus and access to the task panel.

While both ports were well built, stable and pleasant to use, they lacked support for a few standard Python modules which were required for the purposes of developing Koala.

3.1 thread

Neither port supported threads. This meant that the Koala runtime would need to be coded to be single-threaded, and to use `select()` calls to multiplex the various data streams involved, such as keyboard events and incoming RPCs. Sadly, neither port supported `select()`, either — an understandable omission, since the platform itself does not support `select()`.

Psion threading is extremely expensive both in terms of CPU cycles and, as a consequence, of power. The Psion runtime (known as EPOC32) is heavily optimised for single-threaded operation, since only a single application may have control of the display at any one time. In spite of this expectation about the design of Psion applications, the platform does support a fully preemptive thread model, providing a sound base on which to implement the PyThread API.[3]

MoPy now supports the thread and mutex classes natively. Inherently, this allows support for the more convenient threading interface. Performance when no new threads are started is almost identical to that of the threadless ports. Performance once other threads have started suffers slightly more, but the tradeoff is on the whole a favourable one.

[3] The only notable exception was the `ThreadId()` function, which is presently a dirty hack since Psion threads use opaque and transient objects as identifiers, rather than plain integers - the interpreter must therefore make assumptions about the contents of the Epoc ThreadId class in order to obtain a suitable integer value

3.2 socket

The existing ports did not support sockets. The Psion Posix compliance library does provide socket support, making the porting task trivial.

Some noteworthy limitations include:

- The inability of the TCP stack to return a machine name via `gethostbyname()`.
- `getservbyname()` and `SOCK_RDM` are not supported (causing MoPy to fail the standard regression tests).
- The ability to lock the Psion completely (to the level of requiring a soft reset) when using `accept()` calls in particular ways.

Although latter problem is rather severe, it has only been found to be reproducible by forming a local-loopback connection via TCP between two threads in the Python interpreter, and then closing the interpreter forcibly via the System menu. [4] In this situation, the device must be soft-reset using the button in the backup battery compartment.

Socket connections out from the Python interpreter to other processes and via modem to the Internet have not shown any similar problems. Python processes `accept()`ing incoming connections from other processes, such as the Psion 5mx Web application, show similar stability to outgoing connections.

3.3 serial

Each different Operating System platform appears to have its own serial API. Similarly, Python's serial support is splintered between TERMIOS commands under Unix and the `sio` module[4] under Windows, among others.

Because the `sio` and accompanying `serial` modules provide a convenient and extensible interface to the serial ports, their API forms the basis for our `_serial` module on the Psion.

The current `_serial` implementation is extremely basic, supporting a 9600baud, 8-1-N mode, with no flow control. The module exports a single function, `open()`, which is used to open a serial port and to return the corresponding Python object. `_serial` module objects are per-thread, to match the Psion's serial interface semantics — using a `_serial` port object from a thread other than the one in which it was created causes an exception to be raised. To support multi-threaded serial port access, the Python `serial` module is provided as a Psion-specific wrapper to the `_serial` module functionality, providing a more standard, multithreaded port model at the cost of some efficiency.

Clients open ports using the `serial.open(portname)` method, supplying either "ECUART" for the serial cable, or "IRCOMM" for an IrDA[9][10] serial line [5]. Serial port objects currently provide `read()`, `write()` and `close()` methods.

[4] This is sometimes necessary in order to quit a Python script that has entered an endless loop, because no present Python implementation for Psion supports Ctrl-C or similar

[5] Although serial via IrDA is in principle supported, it has not been observed operating between devices as yet

A future implementation will hopefully support fully configurable serial port access, the present choice of settings being a popular lowest-common-denominator used by the PEN prototypes.

4 Conclusion

4.1 Koala Performance

Table 4.1 gives results for the classic string Echo example, on standard RedHat, Windows NT and EPOC32(Psion) platforms, using the Python 1.5.2 interpreter. All values quoted are invocations-per-second.

Platform	TCP Local-loopback	Koala Same-process	Python Native
RedHat Linux2.2.12 450MHz Pentium II	362.7	6072	134700
Windows NT 4 SP6a 450MHz Pentium III	293.9	4717	103400
Psion 5mx, 36MHz ARM 710T CPU	12.19	278.3	8000

The "TCP Local-loopback" result is for method invocations passing between client and server in the same process, via the TCP loopback interface. The "Koala same process" result is for method invocation between client and server in the same process, via Koala, without passing through TCP. The final result, "Python native", comes from calling the Echo servant object directly, bypassing Koala entirely.

Koala is a factor of twenty slower than native access, due solely to the overhead of detecting and locally dispatching the invocations, a process which involves two dictionary lookups and several function calls. Although dictionary lookups are a well-optimised specialty of the standard Python implementation, they still involve hashing and comparison of keys, inevitably introducing a large number of extra CPU cycles as compared to a single function call.

In addition to the CPU-cycle costs incurred in performing method invocations using Koala, invocations performed via TCP loopback are affected by three further factors; the quality of local-loopback in the TCP stack implementation, the cost of thread switching on each platform and the speed with which operands can be marshalled. The results for Psion show these costs are more significant relative to the cost of Koala-internal invocations than under Linux or NT. Memory allocation may make marshalling the extra bottleneck on the Psion, or the quality of its thread switching may have an effect (it is known to be poorly optimised). Resource limits on the smaller device are unlikely to affect performance - the device has 16Mb of RAM but the Python interpreter is limited to using only 2Mb by default, and shows no signs of approaching this limit while running Koala programs.

4.2 MoPy Performance

The expected performance of the MoPy port, based on the capabilities of the processors and operating systems of the test machines, is around a factor of twenty worse than the PC platforms.

Using the pystone benchmark[6] MoPy achieved only 90.22 pystones, as compared to 6060 pystones on the Windows PC platform above.

4.3 Downloads

MoPy is a stable Python 1.5.2 port but lacks some finesse. It passes 34 of the standard Python regression tests, fails 4 (`imageop`, `sha`, `socket`[7], and `time`[8]) and skips 22. Programs not requiring GUI access or Posix-like features such as `select()` support should run unmodified on MoPy.

Some important limitations are:

- The Ctrl-C key sequence cannot be used to interrupt the interpreter. This means that the System menu must be used to kill errant Python processes.

- Like the handles used under Epoc/32 to access the serial ports, Epoc/32 file handles are thread-specific and cannot be used by threads other than the one that created them. When normal files are opened, the Epoc Posix-compliance library against which MoPy is compiled maintains a special Posix server thread which accepts file operations from MoPy threads and maps the file descriptors used internally to the corresponding Epoc/32 file handles. This creates a performance bottleneck but allows multiple threads to share Posix file descriptors.

  Unfortunately, the Posix library does not perform this mapping for the `stdin`, `stdout` and `stderr` file descriptors (descriptors 0, 1 and 2), possibly for performance reasons. As a result, these file descriptors cannot be shared transparently between threads within MoPy. In order to cope with this limitation, the `sys.stdin`, `sys.stdout` and `sys.stderr` file descriptors are specially handled in MoPy, reducing the interpreter's performance on file access. References to file descriptors 0, 1 and 2 are transparently mapped to the corresponding thread-specific standard IO file handle, as a result of which the `os.dup()` call must not be used to duplicate them[9]. Doing so can result in an Epoc/32 KERN-EXEC error in the calling thread.

  A better approach from a performance perspective would have been to replace the default `stdin`, `stdout` and `stderr` objects with custom Epoc-specific versions mapping to the correct thread-specific value. This would avoid the per-operation performance penalty incurred by the current implementation but would

[6] The benchmark was modified to use `time()` instead of `clock()`, since the `clock()` function on the Psion uses microseconds and CLOCKS_PER_SEC is erroneously defined to be 1 in the `types.h` header

[7] See Section 2.3

[8] The basic time functions are available. Localised time functions, such as `timezone` and `daylight` are not.

[9] `os.dup()` duplicates an existing file descriptor to a new file descriptor value, thus potentially bypassing the standard IO workaround

involve a significant re-write of portions of the interpreter. Ideally, the interpreter would be completely re-written to perform all interactions with the operating system via a single special thread within the interpreter, much like the Posix thread used currently, thus avoiding handle/thread interaction issues completely.

- MoPy currently exports the `posix` module. Future versions will rename this module `epoc`, so that poorly supported Posix API calls can be improved or removed.
- Each thread is limited to a 64K stack. All threads share the interpreter's 2Mb heap.

MoPy is available for download from

`http://www.uk.research.att.com/~jnw/downloadables/index.html`

Future improvements will be posted at the above URL.

Koala is a stable system. It is possible to write scripts which import Koala and create one or more ORBs, add custom object interfaces to them and add customised network transports if required. Such scripts can then export standard Python classes implementing the registered interfaces, so that they may be accessed transparently by other Koala-aware scripts.

Koala is not currently available via the web but it will be!

5 Acknowledgements

The authors wish to acknowledge Andy Hopper and Sai-Lai Lo of AT&T Labs Cambridge for their support in this work. Thanks are due to Frank Stajano and Duncan Grisby of AT&T Labs Cambridge and to Diego Lopez de Ipina of the Laboratory for Communications Engineering, Cambridge, for convincing us to give Python a try. Thanks are also due to all the Python contributors, and Guido van Rossum in particular, for producing a language which could be learnt in a single weekend in spite of a killer hangover.

Finally, the AT&T Laboratories Cambridge and the Engineering and Physical Sciences Research Council should be thanked for their funding of this work.

6 Appendix A - Glossary of Terms

- *Object Request Broker*

  The Object Request Broker is the body of software responsible for transparently interfacing between objects running on different programs across a network. The ORB deals with the issues of accepting and unmarshalling incoming Remote Procedure Calls and dispatching them to the relevant objects, and of marshalling and transmitting outgoing RPCs to remote objects.

- *interface*

  An “interface” is a collection of methods and attributes used to define how an object may be manipulated remotely. The OMG’s Interface Definition Language (IDL) provides a set of standard data types which are used when defining interfaces, to ensure compatibility between different CORBA ORB implementations. Interfaces may inherit from each other in a similar manner to C++ or Java classes.

- *servant*

  The term “servant” refers to an object implementing a particular interface and made available through a network by the ORB. The servant is called into by the ORB when an incoming RPC is received, and the results of each invocation are marshalled and returned to the caller transparently by the ORB.

- *stub* A “stub” is an object created by the ORB, through which a remote servant object is accessed. The stub provides the same interface to programs as the servant but marshals the parameters of method invocations and dispatches them to the corresponding servant rather than implementing the method locally.

7 Appendix B - Example IDL

Sample IDL for the follow-me-MP3 example.

```
interface Jukebox {
    typedef sequence<octet> Key;
    typedef unsigned long State;

    Key Reserve(in unsigned long mode);
    void Release(in Key k);

    void Play(in Key k);
    void Stop(in Key k);
    void FFwd(in Key k);
    void RRwd(in Key k);
    void Pause(in Key k);

    attribute State CurrentState;
    attribute string CurrentTrack;
};
```

The Jukebox IDL as supplied to the Koala ORB. The TC module provides TypeCode definitions for the standard OMG IDL datatypes and is assumed to already have been included.

```
class JukeBox:
    """ Interface to a very simple MP3 Jukebox """
```

```
TC_Key = TC.Sequence(TC.Octet)
TC_State = TC.ULong

IDL_Type = "Jukebox"
IDL_Interface = [
                 ("Reserve", (TC.ULong, ), TC_Key),
                 ("Release", (TC_Key,), TC.Void),

                 ("Play", (TC_Key,), TC.Void),
                 ("Stop", (TC_Key,), TC.Void),
                 ("FFwd", (TC_Key,), TC.Void),
                 ("RRwd", (TC_Key,), TC.Void),
                 ("Pause", (TC_Key,), TC.Void),

                 ("_ATTR_CurrentState", (), TC_State),
                 ("_ATTR_CurrentTrack", (), TC.String)
                ]
```

References

[1] Frazer Bennett, David Clarke, Joseph B. Evans, Andy Hopper, Alan Jones, and David Leaske. Piconet - embedded mobile networking. *IEEE Personal Communications, Vol. 4, No. 5, pp 8-15*, October 1997.

[2] Specification of the Bluetooth System Version 1.0A. From `http://www.bluetooth.com/`, July 1999.

[3] Duncan Booth. Python for Epoc Systems. At `http://dales.rmplc.co.uk/Duncan/PyPsion.htm`, August 1999.

[4] Roger Burnham. Win32 Serial Interface Module. From `ftp://ftp.python.org/pub/python/contrib/sio-151.zip` and `http://starship.python.net/crew/roger`.

[5] Otfried Chong. Python for Epoc. At `htpp://www.cs.uu.nl/~otfried/Python/`.

[6] David Evers *(AT&T Laboratories Cambridge)*. Omni Object Services. In preparation.

[7] Fnorb Homepage. At `http://www.fnorb.org/`.

[8] Object Management Group. OMG IDL Syntax and Semantics. In *The Common Object Request Broker : Architecture and Specification, Revision 2.4*, chapter 3. OMG, November 2000.

[9] IrDA Serial Infrared Data Link Standard Specifications. From `http://www.irda.org/`.

[10] IrDA SIR Data Specification. At `http://www.irda.org/standards/pubs/IrData.zip`, February 1999.

[11] Object Management Group. *The Common Object Request Broker: Architecture and Specification*, 2.4 edition, 2000.

[12] omniORB for Python. At `http://www.omniorb.org/omniORBpy/`.

[13] Robert Dunbar Poor. Hyphos : A Self-Organizing Wireless Network. Master's thesis, Massachusetts Institute of Technology, June 1997.

[14] Psion PLC Homepage. At `http://www.psion.co.uk/`.

[15] Jan M. Rabaey, M. Josie Ammer, Julio L. da Silva Jr., Danny Patel, and Shad Roundy. PicoRadio Supports Ad Hoc Ultra-Low Power Wireless Networking. *Wireless Computing - p42-48*, July 2000.

[16] Christian Tismer. Continuations and Stackless Python. In *Proceedings of 8th International Python Conference*, January 2000.

[17] Christian Tismer. Stackless Python. At `http://www.stackless.org/`, March 2000.

[18] Roy Want, Bill N. Schilit, Norman I. Adams, Rich Gold, Karin Peterson, David Goldberg, John R. Ellis, and Mark Weiser. The ParcTab Ubiquitous Computing Experiment. *Mobile Computing*, pages 45–102, 1995.

[19] WAP Formal Specifications. From `http://www.wapforum.org/`, April 1998.