Device and Web Service Objects

Seenu Reddi
ReddiSS at aol dot com

External interface devices extraneous to the main computational capabilities of the computer have always been present ever since the first von Neumann computer was designed back in 1940's. These devices are intended for communicating with the external world and often the human interface for interacting with the computer. Back in 1950's, there were punched card machines to take the input and teletypes for output. Device drivers written specifically to program these devices were in assembly language and it took a skilled person to develop these drivers because of real time constraints and the usually slow nature of the mechanical devices used for realizing these devices.

As the operating systems matured, most of the device drivers got standardized as in Unix/Linux and Windows. Unix initially defined two kinds of drivers, character and block, and these drivers are essentially file based with standard open, read and write procedures. Additional capabilities are supplemented with device ioctl calls which can be used to orient the drivers for their specific idiosyncrasies and operating characteristics. When MS-DOS started, one can see the same nature of drivers being implemented with software interrupts (int 21h) used to accomplish procedure calls. However with the prevalence of internet communications in 1980's, a third driver, network device driver, is added to the Unix/Linux realm since the sockets and other concepts used for realization did not fit within the concept and scope of the other two drivers. One sees the same kind of evolutionary behavior with the Windows operating system. The initial Windows, started out as an add-on to MS-DOS, inherited the primitive non-reentrant, non-shareable device drivers of its parent but had to change considerably when a multi-tasking and mult-user environment was used. This was done through the introduction of VxD drivers, essentially DLL's capable of operating in ring 0 to access the device internals. Windows NT, designed from the bottom up for a multi-tasking and multi-user environment, had an hierarchical view of the devices with I/O request packets traversing up and down to perform device driver operation. The advantage is that the bottom layer represented the devices and the top layer the user. It also accomplished the mapping of user and kernel spaces to this hierarchic layer thus giving the capabilities of shareable devices in a multi-user environments. When Microsoft decided to merge the NT hierarchical architecture with Windows 95/98 ad hoc VxD design, there was confusion and disarray and only way of designing safe drivers is to follow the strict guidelines offered by Microsoft thus stunting the creativity and evolutionary aspects. Microsoft remedied the situation by introducing their latest Windows Driver Framework (WDF) which has an object oriented approach and hides most of the implementation problems with well defined objects and procedures so that the device driver programmer does not have to concern himself with the machine implementation (such as synchronization of resources and power usage) and concentrate only on the specifics related to the device.

The need to accommodate the legacy devices and be compatible with the devices of the previous generations constrains the design environment for developing device drivers. Our objective is not to be constrained by such necessities and develop the framework that affords maximum flexibility and has an object oriented approach for modular, extendible device driver design with bug-free and cross-platform operation. The framework should be able to accommodate new devices with relative ease and exploit their features to their fullest capabilities.

At this time of computer evolution, one can be reasonably certain that the devices in existence at this time, like displays, disk drives, USB, wireless, TCP/IP, .., will continue to be for a few generations in view of the invested capital expenditure in these devices. We will model the framework based on the existing devices and abstract principles to ensure that it will survive even with the onslaught of new devices and technologies.

Device objects are defined using an object oriented approach. We define all kinds of device objects to encompass the existing drivers such as File Objects, Audio Objects, Video Objects, Web Objects, Web Service Objects, USB Objects, TCP/IP Objects, .. so that any device can be mapped to one of these objects easily. Also we define fundamental objects such Synchronization Object, Communication Object, Computational Resource Object, .. so that in principle any computer object can be constructed from these fundamental objects. One of our goals is to realize a distributed computational entity using these objects and web services. A typical example is that Amazon offers a storage entity through their web services and our framework should be able to accommodate this capability through our framework.

Linux distinguishes kernel versus user environment by policy-free and policy-restricted implementation of capabilities (see Rubini, Linux Drivers). The traditional approach has been to differentiate between kernel and user spaces, and give unrestrained capabilities and freedom to kernel objects. Though this has been ingrained in the development and design of operating systems, recent experience with viruses, trojan horses and other unwelcome intrusions that exploit such lack of restraints will question the wisdom of this approach. What we will do is the mark the object as being a kernel or user object, i.e., the object can have an attribute that specifies as whether it has unrestrained kernel capabilities or restrained user capabilities.

Using a web service approach, we define an object as endowed with methods and attributes. Methods are procedures that specify how the capabilities of the underlying object can be used and attributes describe the object itself. The user can extend the object to accommodate for his needs. A simple example is a File Object defined as follows:

```
File Object {
  Method: Open();
  Method: Read();
  Method: Write();
  Method: Close();
```

```
   Attribute: ReadOnly;
   Attribute: MaxSize;
   Attribute: MinSize;
   Attribute: Deletable;
   .
   .
}
```

Objects are defined by XML like constructs. (Note our intent here is expositional and we sacrifice rigor in presentation to convey the basic concepts. Also we need the latitude so as not to be rigid and capable of future expansion for accommodation of ideas unforeseen at this time). Thus we have for the File Object:

```
<File Object>
  <Methods>
    <Open>
    .
    .
    <Close>
  </Methods>
  <Attributes>
    <ReadOnly>
    .
    .
  </Attributes>
</FileObject>
```

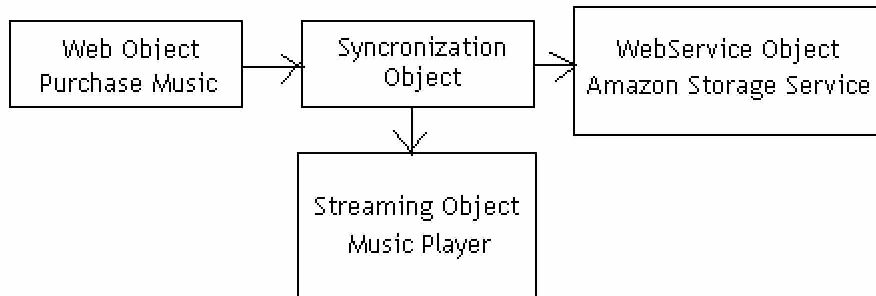
We can also have extended objects like Web Object defined along the following lines to access HTML pages or stream music.

```
Web Object {
   Method: Open();
   Method: Read();
   Method: Write();
   Method: Close();

   Attribute: Streaming;
   Attribute: HTML | MP3 | WMV | ..;
   .
   .
}
```

Similarly a Web Service Object can be defined to access services and resources not locally available or offered by companies specializing in online services. As an example consider the situation where an user wants to buy MP3 music online, charge it to Paypal, save it to Amazon Storage Resource and stream once the entire transaction is completed.

The following diagram succinctly represents the type of objects one can define to accomplish this.



The Synchronization Object synchronizes the various phases involved with the entire operation. The Streaming Object will contain attributes such as TCP/IP or UDP transportation protocol as well real time MP3 (or WMA) streaming capabilities. Note Purchase Music object has to deal with other situations like Paypal rejecting payments and the Synchronization Object should be programmed to handle these other situations.

The Synchronization Object can be realized as a Petri net which can synchronize events and issue new tasks. Real time global computation has been around for a while and the problems involved therein have been researched for the past three decades. Strategies devised to implement asynchronous and deadlock-free computation can be fruitfully integrated into the design of the Synchronization Objects. It has been reported that most of the crashes in the PC can be traced to synchronization primitives and this will be a good lesson in devising synchronization objects that are robust and strable.

The modular device object approach delineated here can be extended to distributed computation using web services as objects and TCP/IP links as communication objects. In such an environment one can conceive services offered by corporations that include computational as well as storage resources and these services can be used to extend the computational power of mobile as well as home/office based computers.