Swapping Stacks and Flooding STREAMS

Mixing Fast Threads with Open Transport

Written by Jonathan 'Wolf' Rentzsch (jon at redshed dot net).

This paper presents an effective method of tying Open Transport to a new thread model that affords both maximum performance and ease of programming. First, a history of multitasking on the Macintosh is presented, and then the new thread model is offered. After a brief history and overview of Open Transport, this paper reviews a chunk of code from a working web server, which illustrates how the two can work together.

Introduction

Way back in 1997, I wrote my first paper for MacHack. It was titled Implementing Threaded IO on the Mac OS, and went into detail about the Device Manager and the File Manager and how to hook them up to the Thread Manager. The paper also showed off the less appealing side of the Thread Manager, and offered advice and code to work around it.

After MacHack 1997, it became clearer about how best to perform threading on lightweight operating systems like the traditional Mac OS. A new scheme was invented, pulsars, which was detailed in a session at MacHack 1998.

This paper pulls together the paper of 1997 with the session of 1998. The examples use Open Transport (instead of the Device Manager and the File Manager of the 1997 paper), and describe in detail how pulsars work. To make all this theory real world, a working multi-threaded web server is included.

Multitasking The SystemTask Way

As introduced in 1984, the original Macintosh could run only one application at a time. The currently running application had the entire Macintosh all to itself. Well, *almost* all to itself. The application had to support smaller parasite programs called **desk accessories**.

These desk accessories were small and quick to open. Folks could use them to quickly take a note, set an alarm or multiply a bunch of numbers. When desk accessories allocated memory, it was stolen from the host application's memory pool. Desk accessories also created windows, and had to be careful to mark them as desk accessory windows so the application wouldn't try to draw into them, close them and other undesirable acts.

Some of these desk accessories needed to draw into their windows at regular intervals. The Alarm Clock desk accessory needed to update the current time, and the Note Pad needed to flash the TextEdit insertion point. Apple handed application developers a special call, `SystemTask`, and instructed them to call it at least 60 times per second.

Behind the scenes, `SystemTask` would walk a list and send an `accRun` message to each desk accessory that requested time. This is illustrated in Figure 1.

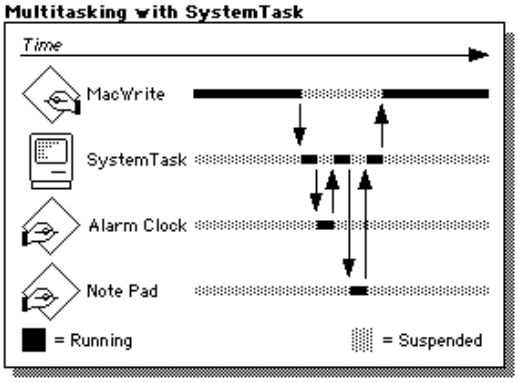

Figure 1.

This is the simplest form of multitasking, and has two distinguishing characteristics. The first is the cooperative nature of the sharing of the processor. The application volunteers to call `SystemTask` so the desk accessories get time to perform periodic tasks. When called, the desk accessories return to `SystemTask` so that other desk accessories and the application get time. If each program doesn't voluntarily relinquish

the processor, the others do not get to proceed.

The second characteristic is that the desk accessories share the stack with the application. When called, the desk accessories are free to push variables and the like onto the stack, however they must pop their items before returning to the application. This is a shame, since the stack is a quick and easy place to store a program's state. Instead, the desk accessory developer would need to explicitly store its state somewhere in the heap, and then find and read that data upon reentry.

Multitasking the MultiFinder Way

Originally an option on System 6, MultiFinder is always active from System 7 forward. As the Macintosh became faster and stuffed with more memory, the copy-quit-launch-paste jig of using multiple applications to accomplish a task became unnecessary. MultiFinder allowed multiple applications to seemingly run at the same time.

Of course, on the single-processor computer, only one application can actually execute at a time. However, computers can quickly switch among different applications to make it appear it is actually performing many tasks at once.

Running many applications at once is very handy, a point not missed by Apple. However, Apple didn't want to rewrite the entire Mac OS, or force application developers to do the same. Instead, in grand programmer tradition, Apple wrote a hack on top of the Mac OS: MultiFinder.

Applications built in the days before MultiFinder thought they had the Mac's memory and processor all to itself. In reality, under MultiFinder, each application would get only a fraction of the Mac's memory and processor's time. Apple needed to design a way to force applications to share both.

The are two basic models to sharing the processor's time: **cooperative** and **preemptive**. The cooperative model relies on applications explicitly yielding control of the processor to each other. This is the model used by applications to give time to desk accessories.

The preemptive model takes away the voluntary system and replaces it with a forced system. Instead of each application deciding when it's ready to yield the processor, the operating system routinely takes control from one application and gives it to another.

Each model has its pros and cons. Most programmers prefer the preemptive model since it frees them about having to worry about when and how often to yield the processor. However, the cooperative model doesn't have the synchronization issues of the preemptive model.

Apple had little choice but to go with the cooperative model. The sad fact is that most Macintosh programming interfaces and large chunks of the Mac OS simply weren't designed to be used by more than one application at once. Put technically, much of the Mac OS wasn't **reentrant**. Preemptive multitasking does a good job of pretending that one processor is many, and the traditional Mac OS simply couldn't handle multiple applications all asking to draw onto the screen at once.

While the cooperative model requires each application explicitly yield the processor, applications written in the days before MultiFinder did no such thing. To make applications share the processor, Apple identified a system call that all applications frequently made, `GetNextEvent`, and modified it to yield the processor in addition to its normal duties.

Now that Apple had applications sharing the processor, they had to find a way for applications to share memory. To accomplish this task, Apple changed the way memory was laid out under MultiFinder. The changes are illustrated in Figure 2.

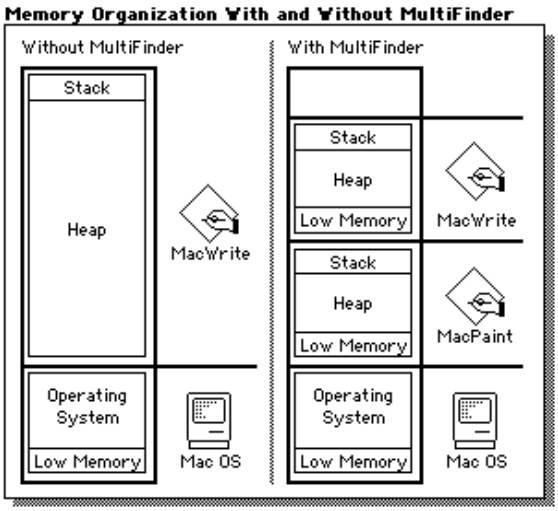

Figure 2.

Apple provided a fairly good programming interface for dynamic (heap) memory allocation, so that aspect of the transition went well. However, applications store information in places other than the heap.

First, the address of the currently executing instruction is stored in the processor's Program Counter. The address of the current location in the application's stack is stored in the processor's Stack Pointer. The processor's registers hold intermediate results of the calculations the application is performing. Environmental information is held in low memory.

Taken together, all these elements define an application in progress. It's an application's **context**.

One of MultiFinder's many jobs were to save an application's context when it was suspended, and restore it before resumption. Figure 3 illustrates the process. Two applications, MacWrite and MacPaint, are running. MacWrite calls `GetNextEvent`, which invokes the Mac OS. The Mac OS looks at its list of running programs and decides who gets to go next. In this case, it's MacPaint. The Mac OS saves MacWrite's context into memory and restores MacPaint's context, and jumps back to where MacPaint left off.

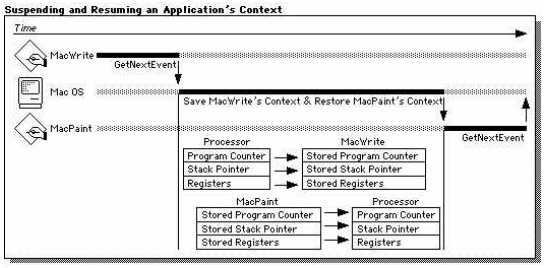

Figure 3.

MacWrite and MacPaint know nothing about being switched in and out. To their perception, their call to `GetNextEvent` simply took a much longer time than before, which bring us to the performance issue.

Multitasking never comes free. Every element of the old context must be copied to a safe location, and every element of the new context must be copied back in. On top of all that work, there's virtual memory. If virtual memory is active and memory is low, then chunks of RAM need to be written to disk and read back in.

To illustrate just how much a performance hit multitasking can be, in the early days of MultiFinder, it was necessary to throttle how often application switching would take place. Otherwise, the Mac would spend all its time switching applications and none on actually getting work done.

Multitasking, in general, slows things down. If you had to run two or more processor-bound applications (that is, applications whose performance is most limited by the speed of the processor), it will always be faster to run each by itself, one after another, than run all of them at once. This is because of the overhead introduced by multitasking.

However, even in this worst-case scenario, multitasking is nice since the computer can still be responsive to the user even while running many applications. Perfect for you guys who calculate pi while surfing the web.

Multitasking truly shines with io-bound applications (applications whose performance is most limited by input/output). While the application is waiting for the hard drive to read a file, other applications can get work done. This benefit is becoming increasingly pronounced

as the performance ratio between processors and IO increases.

Multitasking the Thread Manager Way

Shipped as an extension to System 7 in 1994 and later rolled into the system, the Thread Manager offers a multitasking abilities *within* applications.

A thread is a lightweight execution context. In contrast to the heavyweight application context, a thread only owns two resources: a set of registers and a stack. All other resources are owned by the application, which threads access via simple memory sharing. Figure 4 illustrates the resources owned by a thread, and Figure 5 illustrates the relationship between threads and applications.

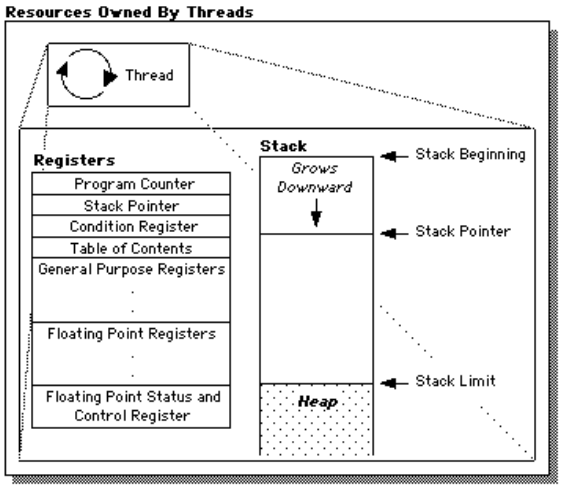

Figure 4.

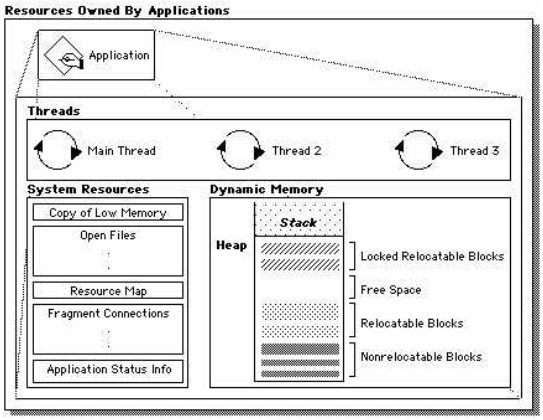

Figure 5.

Threads allow a single application to work on more than one task at once. A web browser is a good candidate for threading. Each window can contain a web page, and the task of reading the HTML text and processing the images are easily separated. Figure 6 suggests such architecture.

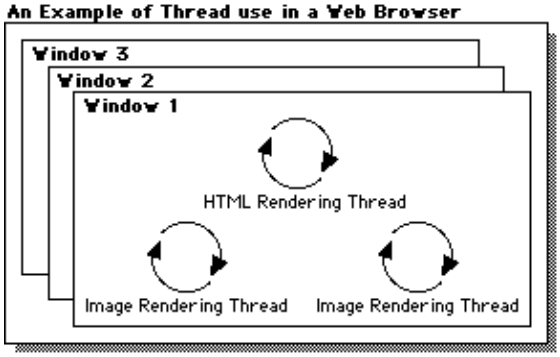

Figure 6.

Initially, the Thread Manager offered two models for thread scheduling: cooperative and preemptive. However, the support for preemptive threads was dropped when the Thread Manager was ported to the PowerPC.

The Thread Manager's cooperative model works much like MultiFinder's cooperative model. Under MultiFinder, each application must call `GetNextEvent` (or its modern equivalent, `WaitNextEvent`), where upon the next application is resumed. Under the Thread Manager, each thread must call `YieldToAnyThread`, where upon the next thread is resumed. The interaction between the Thread Manager and MultiFinder is illustrated in Figure 7. The thin, short vertical arrows represent a thread's call to `YieldToAnyThread`. The thick, long vertical arrows represent an application's call to `GetNextEvent`.

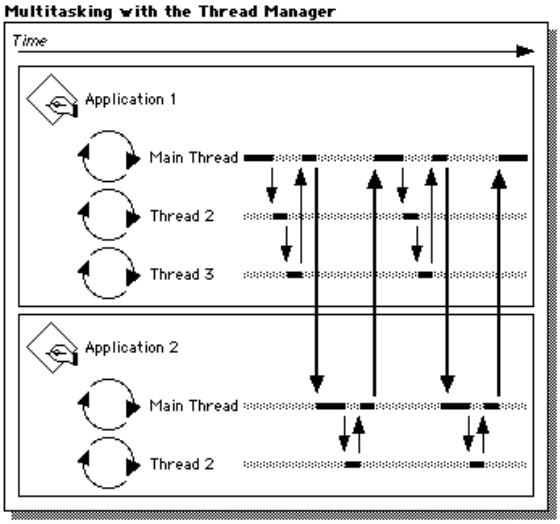

Figure 7.

The paper of 1997 was largely about integrating the Thread Manager with the Device Manager for easy threaded input/output. There are some nice gotchas to be wary of, the biggest of which was the so-called "Window of Death."

The Window of Death is a phenomena where a thread starts a job and goes to sleep, depending on an external event to wake it up. The "window" of the Window of Death refers to the small window of time between when the thread starts the job and when it goes to sleep. If the external event fires before the thread put itself to sleep, the thread is never awoken and sleeps forever.

Different developers took different paths in avoiding the Window of Death. A *develop* article advocated a dual-thread approach. PowerPlant, Metrowerks's C++ framework, used a timer. Other developers used a polling model, where the thread is never put to sleep. The 1997 paper offered XThreads, a package written in C, which avoided the Window of Death and cut scheduling latency. However, all these work-arounds were still hampered by the need to run at SystemTask time.

The traditional Mac OS offers two basic execution levels: SystemTask time (named after the `SystemTask` call) and interrupt time. Most applications spend most of their time at SystemTask time. At SystemTask time, you can call any Toolbox call. However, because of the cooperative nature of MultiFinder, there's no way to predict when an application will get its turn at use the processor.

Interrupt time doesn't suffer from this behavior, however only a small subset of the Toolbox is available at interrupt time (this is because of the reentrancy issue touched on above in the MultiFinder section).

Since the Thread Manager lives at SystemTask time, there is no way to keep threads running when one application is hogging the processor. And before you start cursing about poorly written applications that hog the processor, keep in mind that even the best written application will monopolize the processor while the user has the mouse button down.

Multitasking the Pulsar Way

Pulsars is a model that provides threading at interrupt time with very little latency. The best way to explain the pulsar model is to contrast it with the Thread Manager. As an example, we'll copy a file. The main task of copying a file is repetitively reading a block from the source file and writing it to the destination file.

Listing 1 presents some C-style psuedocode, which copies a file. For simplicity's sake, the code that avoids the Window of Death is not shown, so this code is not suitable as a model for a real implementation.

Listing 1.

```
   CopyFileViaThreadManager( File
sourceFile, Folder destinationFolder )
   {
     /*  Create the destination file in the
destination folder.*/
     sourceFileName = GetFileName(
sourceFile );
     destinationFile = CreateFile(
sourceFileName, destinationFolder );

     /*  Open both files.*/
     sourceFileRef = OpenFile( sourceFile
);
     destinationFileRef = OpenFile(
destinationFile );

     /*  Set the destination file to the
source file's size.*/
     sourceFileSize = GetFileSize(
sourceFileRef );
     SetFileSize( destinationFileRef,
sourceFileSize );

     /*  Repetitively read from the source
file and write to the destination file.*/
     while( sourceFileSize > 0 ) {
       block = ReadFile( sourceFileRef,
BLOCKSIZE, ThreadManagerCallback );
      SetThreadState( kCurrentThreadID,
kStoppedThreadState, kNoThreadID );
```

```
      WriteFile( destinationFileRef,
block, BLOCKSIZE, ThreadManagerCallback );
      SetThreadState( kCurrentThreadID,
kStoppedThreadState, kNoThreadID );

      sourceFileSize -= BLOCKSIZE;
    }

    /*  Close both files.*/
    CloseFile( destinationFile );
    CloseFile( sourceFile );
  }

  ThreadManagerCallback( ThreadID thread )
  {
    SetThreadState( thread,
kReadyThreadState, kNoThreadID );
  }
```

As shown in Listing 1, our copying engine calls `ReadFile` and then goes to sleep by calling `YieldToAnyThread`. When the read is complete, the Mac OS calls the supplied callback `ThreadManagerCallback` at interrupt time. Since you can't use much of the thread manager at interrupt time, all we can do is mark the thread as ready and return. Figure 8 is a visualization of this technique.

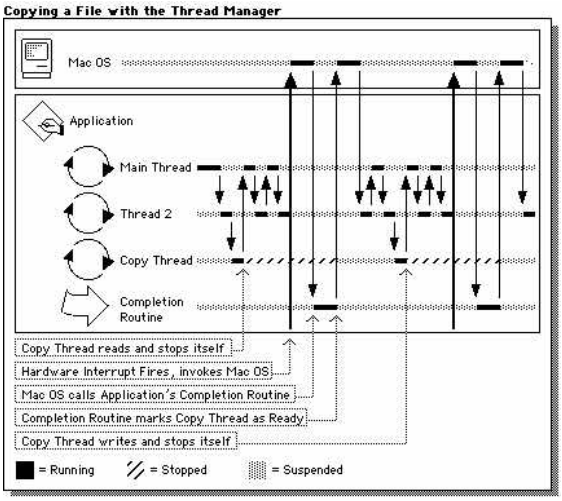

Figure 8.

Unfortunately, this technique is subject to great latency. The time between when the completion routine marks a thread as ready and when the thread runs is large, and can vary widely. The XThreads package cut down on the internal application latency by maintaining a priority thread queue. However, since the Thread Manager works at SystemTask time, our application will still be held up by something as innocent as the user holding down the mouse button.

Listing 2 exhibits the pulsar model. As you can see, the code for the pulsar model is almost exactly like the Thread Manager model, however the pulsar model acts very differently from the Thread Manager model.

Listing 2.

```
  CopyFileViaPulsar( File sourceFile,
Folder destinationFolder )
  {
    /*  Create the destination file in the
destination folder.*/
    sourceFileName = GetFileName(
sourceFile );
    destinationFile = CreateFile(
sourceFileName, destinationFolder );

    /*  Open both files.*/
    sourceFileRef = OpenFile( sourceFile
);
    destinationFileRef = OpenFile(
destinationFile );

    /*  Set the destination file to the
source file's size.*/
    sourceFileSize = GetFileSize(
sourceFileRef );
    SetFileSize( destinationFileRef,
sourceFileSize );

    /*  Repetitively read from the source
file and write to the destination file.*/
    while( sourceFileSize > 0 ) {
      block = ReadFile( sourceFileRef,
BLOCKSIZE, PulsarCallback );
      Sleep();  /*  Different.*/

      WriteFile( destinationFileRef,
block, BLOCKSIZE, PulsarCallback );
      Sleep();  /*  Different.*/

      sourceFileSize -= BLOCKSIZE;
    }

    /*  Close both files.*/
    CloseFile( destinationFile );
    CloseFile( sourceFile );
  }

  PulsarCallback( Thread thread )
  {
    Pulse( thread );  /*  Different.*/
  }
```

Instead of merely marking the thread as ready from the completion routine (`PulsarCallback`), the completion routine actually saves the caller's context and swaps in `CopyFileViaPulsar`'s context and reenters it at interrupt time. The Copy Thread performs the block write and immediately returns to the completion routine via the `Sleep` call, which then returns to the Mac OS.

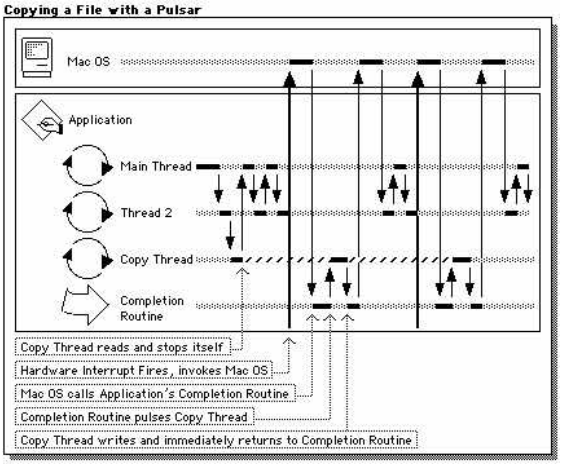

Figure 9.

The completion routine passes control to the Copy Thread, depending on the Copy Thread to quickly pass control back. In effect, the completion routine quickly starts and stops the Copy Thread, or "pulses" it. This is where the model's name originates.

A pulsar is simply a callback that starts and stops a thread from within the callback. Since the Mac OS makes great use of callbacks, the model is a great match.

Since the Mac OS supplies callbacks that interrupt the current process (like VBL tasks, Time Manager tasks and Deferred Tasks), you can architect your application to perform much of its work at interrupt time, effectively emulating a preemptive operating system. Your application can continue working even if the user is holding down the mouse button in someone else's application. However, the pulsar model does require threads make explicit yielding calls, so that benefit of preemptive multitasking isn't fully realized.

But enough with all this threading, let's dive into Open Transport.

A Brief History of Open Transport

The very first Macintosh shipped with built-in networking in 1984. Back then, AppleTalk over LocalTalk was a "medium speed" networking technology, topping out a 230Kbps. Eventually Apple added support for faster network hardware (Ethernet, Token Ring) and different network protocols (IP, IPX).

After the Power Macs shipped, Apple could have simply ported the existing network software. However, it wouldn't be an easy port: much of the software was written in unportable 68K assembly language. Additionally, Apple's support of the increasingly important Internet Protocol was provided via MacTCP - a low-performance hack that was entirely Apple-proprietary.

Apple made the decision to move to fast, flexible and standards-based network software: STREAMS with a X/Open Transport Interface.

STREAMS (uppercased to help separate from the overloaded stream concept), was first introduced by AT&T in their UNIX System V. Dennis M. Ritchie (yes, **that** Dennis Ritchie, co-inventor of UNIX and C) wanted a better architecture for performing input/output.

Following the UNIX philosophy of tying small tools together to accomplish large tasks, he devised a flexible method of layering modules on top of each other. This was an interesting take on a networking architecture. Traditionally, the networking software provided with an operating system was monolithic: in order to add a new protocol, you'd need to rewrite and recompile at least a library, if not the kernel itself. STREAMS made it easy to add new protocols and devices at will.

By the time Apple was in the market for new networking software, STREAMS had grown up. At first, STREAMS's flexibility made it slower then other operating system's monolithic architecture. However, a small company named Mentat solved the performance problems and proceeded to license their product, Mentat Portable Streams (MPS), to numerous Operating System suppliers including Sun (Solaris), Digital (Ultrix) and Hewlett-Packard (HP/UX).

MPS was a great fit for Apple, as STREAMS allows multiple protocols simultaneously. While TCP/IP was coming on strong, Apple still needed to support AppleTalk. Furthermore, all of Mentat's code was in portable C. While some modifications were necessary to move MPS from a UNIX-style Operating System to the

Mac OS, it was much faster than writing from scratch.

To add icing to the cake, MPS supports the X/Open Transport Interface (XTI). X/Open is a standards group formed a while back. One of their standards is XTI, a network programming interface. In theory, this would make it easier to move code from Unix to the Mac. In reality, most Unix network programming is done with BSD-style sockets. While Mentat offers a sockets API for MPS, Apple chose not to license it. However, Matthias Neeracher wrote and maintains GUSI, a free open-source sockets interface that rides on top of Open Transport.

Open Transport's Architecture

In the previous section, I glossed over Open Transport's architecture. Now we get down and dirty.

Perhaps you've heard of "protocol stacks". The basic observation is that protocols can be built two ways: as one large protocol, or as many smaller protocols working together. Just as modular software gives you more flexibility, modular protocols offer better support for different tasks and future extension. Most of today's networking protocols are "stacked", that is, each protocol relies of the services of a lower protocol until you reach the hardware. The basic model is illustrated in Figure 10.

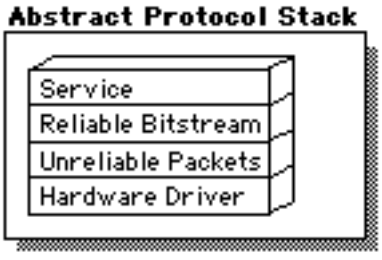

Figure 10.

The most famous protocol stack is the Open Systems Interconnection (OSI) model. OSI attempts to define a generic protocol stack as a model for others to implement. See Figure 11.

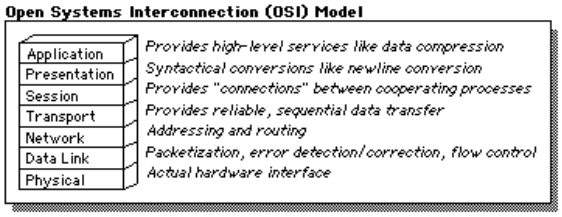

Figure 11.

While the OSI model is largely academic, protocols *have* been built with a one-to-one correspondence with OSI.

Let's look at a few of real-world examples of protocol stacks. The first is when you mount a disk from another Macintosh using Personal File Sharing over LocalTalk. See Figure 12.

Figure 12.

Our next example is when you download a web page over an Ethernet network, which is tied to the Internet. See Figure 13.

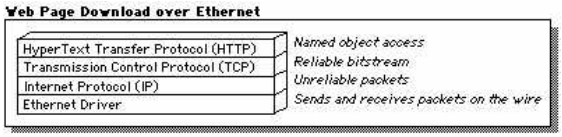

Figure 13.

Finally, Figure 14 shows an example of downloading a web page over a modem connection:

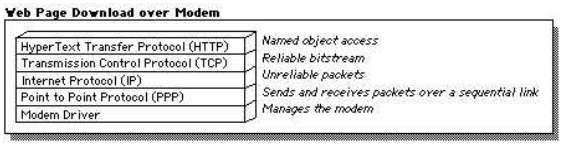

Figure 14.

As you can see, each module uses the services of the one below it. Open Transport is simply a realization of this simple abstraction.

Building Stacks

Now we're ready to put some of this architectural theory into practice. Let's say we're writing a simple program that downloads web pages. The protocol for

downloading web pages is HTTP, which rides upon TCP, which rides upon IP, which rides upon Ethernet.

We want to tell Open Transport to build this stack for us. First, we create a "configuration". A configuration is a black box that describes a protocol stack to Open Transport. We call the `OTCreateConfiguration` function, which takes a C string as its sole parameter and returns an `OTConfigurationRef`. The C string is a comma-delimited string describing the stack we want.

While you can specify the stack to build in complete detail (for example, "tcp, ipm, enet0" or "afp, asp, ddp, enet0"), this is inflexible. The user may decide to connect via a LocalTalk port, Ethernet port or Modem port. The better way is to only specify the top-most protocol you want (for example "tcp" or "afp") and allow Open Transport to figure out how to build the stack for you.

You might wonder how Open Transport figures out how to build the stack. While Open Transport could just read a configuration file, instead Open Transport employs a modular approach. It will call upon user-installed modules that return how to build the stack. These modules are known as **configurators**. A third party could write a configurator that automatically pushes their module on top of any IP module. This third party module would then witness all Internet traffic as it passes by, which is tremendously flexible.

Given an `OTConfigurationRef` returned by `OTCreateConfiguration`, we can then open an **endpoint**. From the user's view, an endpoint is an instance of a stack of modules. There are four functions that can open an endpoint: `OTOpenEndpoint`, `OTAsyncOpenEndpoint`, `OTOpenEndpointInContext` and `OTAsyncOpenEndpointInContext`.

The first two, `OTOpenEndpoint` and `OTAsyncOpenEndpoint`, were the original methods to open an endpoint in synchronous/blocking mode and asynchronous/nonblocking mode, respectively (I'll explain what those mean later on).

Under Carbon, those functions were replaced with `OTOpenEndpointInContext` and `OTAsyncOpenEndpointInContext`. These operate largely the same as the original two, except they take an additional `OTClientContextPtr` parameter. This makes it easier for Open Transport to track which resources belong to which client.

If you're writing a standard application, you can simply pass `nil` as the `OTClientContextPtr` parameter and Open Transport will be happy. Indeed, when compiling for Carbon, Apple includes a macro version of `OTOpenEndpoint` and `OTAsyncOpenEndpoint`, which call through to `OTOpenEndpointInContext` and `OTAsyncOpenEndpointInContext`, supplying `nil` in the `OTClientContextPtr` parameter.

To close the endpoint, you call `OTCloseProvider`, passing the `EndpointRef` you were handed when you opened the endpoint.

One more thing while we're talking about endpoints: an endpoint can have a callback associated with it. This callback is known as a **notifier** and is used by Open Transport to send you messages about what is going on.

Listing 3 presents a simple function that shows off the entire Open Transport lifecycle for an application.

Listing 3.

```
#if  TARGET_API_MAC_CARBON
  #define  InitOpenTransport()    \
    InitOpenTransportInContext(   \
      kInitOTForApplicationMask, nil )
  #define  CloseOpenTransport()  \
    CloseOpenTransportInContext( nil )
#endif

  OSStatus
OpenAndClose()
{
  Boolean             initedOT = false;
  OTConfigurationRef  cfig;
  EndpointRef         ref =
kOTInvalidEndpointRef;
  OSStatus            err, err2;

  err = InitOpenTransport();

  if( !err )
    initedOT = true;

  if( !err ) {
    cfig = OTCreateConfiguration( "tcp"
);
    ref = OTOpenEndpoint( cfig, 0, nil,
&err );
  }

  if( ref != kOTInvalidEndpointRef ) {
    err2 = OTCloseProvider( ref );
    if( !err )
      err = err2;
```

```
    }

    if( initedOT ) {
      CloseOpenTransport();
    }

    return( err );
  }
```

First thing we notice is a couple of macros that are only defined if we're compiling for Carbon. Much like the `OTOpenEndpoint` / `OTOpenEndpointInContext` story above, Apple took the simple parameterless `InitOpenTransport` and replaced it with the two-parameter `InitOpenTransportInContext`. Same deal with `CloseOpenTransport` versus `CloseOpenTransportInContext`.

While Apple provided a convenient macro to map the old `OTOpenEndpoint` onto the new `OTOpenEndpointInContext`, they didn't provide the same for `InitOpenTransport` and `CloseOpenTransport`. So, we do so here.

After declaring a few variables, we call `InitOpenTransport` (which, under Carbon, is a macro that expands to `InitOpenTransportInContext`). If successful, we call `OTCreateConfiguration`, which builds a description of the stack to build and passes back a pointer to the description. If `OTCreateConfiguration` fails, it will pass back an invalid pointer. You'll notice that we blindly pass the result to `OTOpenEndpoint`. This is explicitly documented as being okay: `OTOpenEndpoint` will make sure the configuration is valid before using it. Otherwise, `OTOpenEndpoint` will return an appropriate error code.

Since this is only an demonstration, we don't do anything with the endpoint -- we simply close it with `OTCloseProvider` and go on our merry way.

Finally, if we successfully initialized Open Transport, we close it here with a call to `CloseOpenTransport` (which, under Carbon, is a macro that expands to `CloseOpenTransportInContext`).

The Open Transport Programming Modes

Open Transport offers two binary modes (synchronous / asynchronous and blocking / nonblocking), which combined offer four distinct modes of operation. As we'll see later, only three of these modes make sense.

The first mode, synchronous / asynchronous, determines how Open Transport reacts when called from you application. In synchronous mode, your application is halted while Open Transport executes your request. This is similar to using the synchronous version of the File Manager calls (i.e. `PBReadSync`), or setting the `sendMode` of `AESend` to `kAEWaitReply`.

In asynchronous mode, Open Transport simply queues your request and returns immediately to your application. This is similar to using the asynchronous version the File Manager calls (i.e. `PBReadAsync`), or setting the `sendMode` of `AESend` to `kAEQueueReply`.

The second mode, blocking / nonblocking, determines how Open Transport reacts when it must wait to send or receive data. In blocking mode, if a call is made synchronously, Open Transport simply waits as long as necessary for the data to be sent or received. On the other hand, if the call was made asynchronously, Open Transport will return an error if it cannot immediately execute the request. In nonblocking mode, Open Transport attempts to send or receive the data and returns an error if it cannot immediately execute the request.

The purpose of all these modes can be confusing. To clarify, let's take a sample Open Transport call and see how it's affected by the different modes. The sample call we'll use is `OTConnect`, which is the call responsible for creating a connection between your computer and a remote computer.

Synchronous/Blocking: In this mode, calling `OTConnect` would effectively hang the entire Mac until the connection was established. Basically, Open Transport

would sit in a tight loop babysitting the connection until completion.

While waiting for a connection doesn't seem that bad, keep in mind that the user sometime mistypes an address or the network goes down. In this situation, it can take Open Transport up to four minutes to give up and return an error. However, your application will never see that error since the user would have pulled the plug after the first 30 seconds.

However, you can have Open Transport call back into your application while it babysits the connection. From your callback, you can safely call `YieldToAnyThread` and `WaitNextEvent`, giving other threads and applications time. In order to have Open Transport perform this act of kindness, you must call `OTUseSyncIdleEvents` before performing any potentially lengthy tasks. When turned on, Open Transport will send a special event `kOTSyncIdleEvent` to your endpoint's notifier callback.

In summary, the synchronous/blocking mode only makes sense if you couple it with the Thread Manager by calling `OTUseSyncIdleEvents`. This model is very easy to program, however suffers from mediocre performance since it depends on other applications calling `WaitNextEvent` in a timely fashion.

Synchronous/Nonblocking: This mode is very similar to the synchronous/blocking mode, except for how it handles congestion when sending and receiving data. Under synchronous/blocking, if your computer has to wait for the remote side to catch up, Open Transport does the waiting for you. Under this mode, Open Transport doesn't wait -- it returns an error. Once the remote side has caught up, Open Transport will send you a `T_GODATA` or `T_GOEXDATA` message to your endpoint's notifier callback.

Asynchronous/Blocking: This mode is the mode of professionals. It's difficult to program but as fast as you can get. In this mode, `OTConnect` would start the connection process in the background and immediately return to your application. You are free to do anything you want while waiting for Open Transport to complete your request. A couple of suggestions include displaying a progress bar, and giving time to other threads and applications.

When Open Transport has completed the connection, it calls your endpoint's notifier at interrupt time with the event `T_CONNECT`.

Unlike synchronous/blocking and synchronous/nonblocking, you cannot efficiently tie this model to the Thread Manager since your notifier callback will be called at interrupt time. However, the pulsar model works at interrupt time, and makes a great match.

Asynchronous/Nonblocking: This is the useless mode as mentioned above. It provides no benefit over the asynchronous/blocking model, and requires extra code on your part to handle resource exhaustion.

Writing a Threaded Open Transport Server

To illustrate the differences between the various models (synchronous/blocking with the Thread Manager, asynchronous/blocking with notifiers and asynchronous/blocking with pulsars), we'll code up the same task in each of the three models. The task is to create a listener endpoint on port 80 -- the port used by the HyperText Transfer Protocol (HTTP, the protocol of web servers).

First up is synchronous/blocking with the Thread Manager, as illustrated in Listing 4.

Listing 4.

```
pascal
  void*
ThreadedListener(
  void  */*param*/ )
{
  OSStatus       err, err2;
  TEndpointInfo  info;
  EndpointRef    ref = OTOpenEndpoint(
OTCreateConfiguration( "tilisten,tcp" ), 0,
&info, &err );

  if( !err )
    //  Register for quit messages.
    err = RegisterQuitting(
QuitOTThread, ref, nil );
  if( !err )
    //  OTOpenEndpoint creates a
    //  synchronous/nonblocking
    //  endpoint. Set it to
    //  synchronous/blocking.
    err = OTSetBlocking( ref );
  if( !err )
    //  Install our yielding notifier.
    err = OTInstallNotifier( ref,
```

```
gThreadedNotifier, nil );
     if( !err )
       //  Tell Open Transport to call it
       //  when waiting.
       err = OTUseSyncIdleEvents( ref, true
);
     if( !err ) {
       //  Enable IP address reuse.
       TOption   option = { sizeof option,
INET_IP, IP_REUSEADDR, 0, true };
       TOptMgmt  request = { sizeof option,
sizeof option, (UInt8*) &option,
T_NEGOTIATE };
       err = OTOptionManagement( ref,
&request, &request );
       if( !err && option.status !=
T_SUCCESS ) {
         err = option.status;
       }
     }
     if( !err ) {
       //  Bind to port 80.
       InetAddress  in = { AF_INET, 80, 0,
0,0,0,0,0,0,0,0 };
       TBind    bind = { sizeof in, sizeof
in, (UInt8*) &in, 1 };
       err = OTBind( ref, &bind, nil );
     }

     bool  done = false;
     while( !err && !done ) {
       //  Wait for a connection.
       InetAddress  address;
       TCall    call = { sizeof address,
sizeof address, (UInt8*) &address, 0, 0, 0,
0, 0, 0, 0 };
       err = OTListen( ref, &call );

       if( !err ) {
         //  ...Hand off connection to
         //  worker thread...
       }
     }

     //  Unregister for quit messages.
     err2 = UnregisterQuitting(
QuitOTThread, ref, nil );
     if( !err )
       err = err2;

     //  Close the endpoint.
     if( ref != kOTInvalidEndpointRef ) {
       err2 = OTCloseProvider( ref );
       if( !err )
         err = err2;
       ref = kOTInvalidEndpointRef;
     }

     return( nil );
   }
```

First we create a configuration using `OTCreateConfiguration`, in this case `tilisten, tcp`. For clients, it's common to pass `kTCPName` as the parameter to `OTCreateConfiguration`.

Note we push an extra module on top of the TCP module: `tilisten`. Open Transport makes it very difficult to have a server deal with multiple simultaneous connection requests. *Inside Macintosh: Networking with Open Transport* puts it this way:

1. *You have a listening endpoint (one bound with a qlen greater than 0) in asynchronous mode. (The problem is independent of the mode of the listening endpoint but, for the sake of this example, we'll assume the listening endpoint is in asynchronous mode.)*

2. *An incoming connection arrives, and the listening endpoint calls your notifer with a `T_LISTEN` event.*

3. *Your notifer reads the details of the incoming connection using the `OTListen` routine.*

4. *Your notifer decides to accept the incoming connection by calling the function `OTAccept`.*

5. *However, the `OTAccept` call fails with a `kOTLookErr` because there is another pending `T_LISTEN` event on the listening endpoint. (This behavior is explicitly allowed in the XTI specifcation.)*

`tilisten`, first shipped with Open Transport 1.1.1, handles this mess for you -- all you have to do is push it on top of the TCP module for your listening endpoint.

Now that `OTCreateConfiguration` created the configuration, we use `OTOpenEndpoint` to open an endpoint that listens for requests. But before we start listening, we have to set up a few things. First off, we register our threads for quit messages. This is my own code from my Quitting Package, which is included on the MacHack CD.

It turns out that `OTOpenEndpoint` creates an endpoint in synchronous/nonblocking mode, so we need to change it to synchronous/blocking using `OTSetBlocking`. Next up we install our yielding notifier, which looks like Listing 5.

Listing 5.

```
  pascal
  void
ThreadedNotifier(
  void         */*context*/,
  OTEventCode  code,
```

```
  OTResult     /*result*/,
  void         */*cookie*/ )
{
  if( code == kOTSyncIdleEvent ) {
    YieldToAnyThread();
  }
}
```

`ThreadedNotifier` is called repeatedly by Open Transport while it waits for a task to complete. To enable this useful behavior, you must call `OTUseSyncIdleEvents`.

The next chunk of code works around a design issue of TCP/IP. It turns out TCP/IP forces a two minute wait between disconnection and reconnection of a port. That means if the user quits and immediately relaunches our application, our server would be deaf for two minutes. This code works around this issue by sending a message directly to the IP module sitting below TCP (`OTOptionManagement`).

Now we've set everything up the way we need it. All we have to do now is tell Open Transport which port to listen in on (in this case, port 80) by calling `OTBind` and then wait for connections by calling `OTListen`.

Now we'll do the same thing using the asynchronous/blocking with notifiers, except instead of spinning off a worker thread, we'll have the listener handle the request itself. This is to better illustrate the flow of messages a notifier sees in the course of handling an connection. See Listing 6.

Listing 6.

```
pascal
  void
MyNotifier(
  void         */*context*/,
  OTEventCode  code,
  OTResult     result,
  void         *cookie )
{
  OSStatus  err = noErr;

  switch( code ) {
    case T_OPENCOMPLETE: {
      gRef = (EndpointRef) cookie;

      err = OTSetBlocking( gRef );

      if( !err ) {
        TOption    option = { sizeof(
option ), INET_IP, IP_REUSEADDR, 0, true };
        TOptMgmt  request = { sizeof
option, sizeof option, (UInt8*) &option,
T_NEGOTIATE };
        err = OTOptionManagement(
(EndpointRef) cookie, &request, &request );
      }
      /*  Continued at
T_OPTMGMTCOMPLETE...*/
      } break;
    case T_OPTMGMTCOMPLETE: {
      InetAddress  in = { AF_INET, 80,
0, 0,0,0,0,0,0,0,0 };
      TBind    bind = { sizeof in,
sizeof in, (UInt8*) &in, 20 };
      err = OTBind( gRef, &bind, nil );
      /*  Continued at
T_BINDCOMPLETE...*/
      } break;
    case T_BINDCOMPLETE: {
      /*  Continued at T_LISTEN...*/
      } break;
    case T_LISTEN: {
      /*  ...Worker accepts the
connection...*/
      InetAddress  address;
      TCall    call = { sizeof address,
sizeof address, (UInt8*) &address, 0, 0, 0,
0, 0, 0, 0 };
      err = OTListen( gRef, &call );

      if( !err )
        err = OTAccept( gRef, gRef,
&call );
      /*  Continued at
T_ACCEPTCOMPLETE...*/
      } break;
    case T_ACCEPTCOMPLETE: {
      /*  Continued at T_PASSCON...*/
      } break;
    case T_PASSCON: {
      OTSnd( gRef, "hello", sizeof(
"hello" ) - 1, 0 );

      err = OTSndOrderlyDisconnect( gRef
);
      /*  Continued at T_ORDREL...*/
      } break;
    case T_ORDREL: {
      err = OTRcvOrderlyDisconnect( gRef
);
      /*  Continued at T_LISTEN...*/
      } break;
  }
}
```

Most of the code to handle the connection is within the notifier, `MyNotifier`. However, keep in mind notifiers are called by Open Transport, which means we need to kick off Open Transport before it can call our notifier.

You can begin the process by calling `OTAsyncOpenEndpoint` as illustrated in Listing 7. `OTAsyncOpenEndpoint` is similar to `OTOpenEndpoint` as they both create an endpoint based on a `OTConfigurationRef` generated by `OTCreateConfiguration`. However, they differ in three ways. First, `OTAsyncOpenEndpoint` creates an endpoint in asynchronous/nonblocking mode. Second, `OTAsyncOpenEndpoint` takes a notifier as an parameter -- there's no need to call `OTInstallNotifier`. Finally, `OTAsyncOpenEndpoint` immediately returns to its caller. Later on, a `T_OPENCOMPLETE` will be send to the newly opened endpoint's notifier.

Listing 7.

```
OTAsyncOpenEndpoint(
OTCreateConfiguration( "tilisten,tcp" ), 0,
nil, NewOTNotifyUPP( MyNotifier ), nil );
```

Take a look at `MyNotifier` and notice how it accepts four parameters. The first parameter, `context`, is for your own use. Think of it as a reference constant. The second parameter, `code`, indicates what event Open Transport is reporting to your notifer. You'll notice we switch off this code to figure out what to do. The third parameter, `result`, contains the error code of the event. The final parameter, `cookie`, contains event-specific information. For example, during a `T_OPENCOMPLETE` event, `cookie` holds the newly created endpoint.

Initially, `MyNotifier` receives a `T_OPENCOMPLETE` event. It stores away the `EndpointRef` stored away in `cookie` -- we won't see it again. First we change the mode from asynchronous / nonblocking to asynchronous / blocking by calling `OTSetBlocking`. Then we sidestep the TCP two minute delay by calling `OTOptionManagement`. Note that `OTOptionManagement` operates asynchronously, so we return from our notifier and await for Open Transport to call our notifier with the `T_OPTMGMTCOMPLETE` event.

From there we chain to binding the endpoint (`OTBind`) and listening for connections (`OTListen`). For this example, we simply send a friendly "hello" to the incoming connection (`OTSnd`), and disconnect them (`OTSndOrderlyDisconnect` and `OTRcvOrderlyDisconnect`).

Finally, let's see pulsars in action. Unfortunately, this paper described the pulsar model, not an implementation. In order to provide a sample of pulsars in action, I'll have to use my commercial implementation of pulsars: Red Shed Threads. Listing 8 shows how a C++ object named `Listener` handles the task.

Listing 8.

```
void
Listener::Entry()
{
  EndpointRef  ref =
kOTInvalidEndpointRef;
    bool         done = false;
    OSStatus     err =
RegisterForQuitMessages( &this-
>messageQueue );

    if( !err ) {
      ref = TOTOpenEndpoint( this,
OTCreateConfiguration( "tilisten,tcp" ), 0,
nil, &err, 0 );
    }

    if( !err ) {
      //  Enable IP address reuse.
      TOption   option = { sizeof( option
), INET_IP, IP_REUSEADDR, 0, true };
      TOptMgmt  request = { sizeof option,
sizeof option, (UInt8*) &option,
T_NEGOTIATE };
      err = TOTOptionManagement( this,
ref, &request, &request, 0 );
      if( !err && option.status !=
T_SUCCESS ) {
        err = option.status;
      }
    }

    if( !err ) {
      //  Bind to port 80.
      InetAddress  in = { AF_INET, 80, 0,
0,0,0,0,0,0,0 };
      TBind        bind = { sizeof in,
sizeof in, (UInt8*) &in, 20 };
      err = TOTBind( this, ref, &bind,
nil, 0 );
    }

    while( !err && !done ) {
      OTEventCode    otEvent;
      AtomicMessage  message =
WaitReceiveRedShedThreadMessage( this, nil,
kNoAtomicMessage, (long*) &otEvent, nil,
nil );
      switch( message ) {
        case kQuitMessage:
          done = true;
          break;
        case kOTEventMessage:
          switch( otEvent ) {
            case T_LISTEN:
              err =
Server::ReceiveConnection( ref );
              if( err ) {
                InetAddress  a;
                TCall  call = { sizeof a,
sizeof a, (UInt8*) &a, 0, 0, 0, 0, 0, 0, 0
};
                err = OTListen( ref, &call
);
                if( !err )
                  err = TOTSndDisconnect(
this, ref, &call, 0 );
              }
              break;
            case T_ACCEPTCOMPLETE:
            case T_DISCONNECTCOMPLETE:
              break;
          }
          break;
      }
    }

    //  Kill our endpoint.
    if( ref != kOTInvalidEndpointRef ) {
      err = OTSetSynchronous( ref );
      err = OTCloseProvider( ref );
      ref = kOTInvalidEndpointRef;
    }

    //  Drop down to Event Task time.
    WaitEventTask( this, kPriority );
    UnregisterForQuitMessages(
&this>messageQueue );
  }
```

As you can see, the asynchronous/blocking with pulsars code looks much like the synchronous/blocking with the Thread Manager. While similar, there are differences.

First off, note that all the asynchronous Open Transport functions (`OTAsyncOpenEndpoint`, `OTOptionManagement`, `OTBind`, etc.) have been replaced with similarly named wrapper functions (`TOTOpenEndpoint`, `TOTOptionManagement`, `TOTBind`, etc.). That "T" in front of each function stands for "Threaded".

To thread an asynchronous Open Transport function, you must first call the Open Transport function and then wait for Open Transport to call your notifier with the correct event code. Between when your thread calls Open Transport and when Open Transport calls your notifier, your thread can sleep.

The threaded wrapper functions provide this code for you. Let's take `TOTOpenEndpoint` for example, see Listing 9.

Listing 9.

```
  EndpointRef
TOTOpenEndpoint(
  RedShedThread       *thread,
  OTConfigurationRef  cfig,
  OTOpenFlags         oflag,
  TEndpointInfo       *info,
  OSStatus            *err,
  long                patience )
{
  OSStatus  err2;
  EndpointRef  result =
kOTInvalidEndpointRef;

    err2 = OTAsyncOpenEndpoint( cfig,
oflag, info, NotifyRedShedThread, thread );

    if( !err2 ) {
      OTEventCode  event = T_OPENCOMPLETE;
      err2 = WaitOTEvent( thread, &event,
(long*) &result, patience );
      if( err2 )
        result = kOTInvalidEndpointRef;
    }
    if( !err2 )
      err2 = OTSetBlocking( result );

    if( err )
      *err = err2;
    return( result );
}
```

First `TOTOpenEndpoint` calls upon `OTAsyncOpenEndpoint`. Then it waits for Open Transport to issue the `T_OPENCOMPLETE` event.

The final point of difference between the models is how the `T_LISTEN` event is handled. The synchronous/blocking Thread Manager model calls OTListen, which blocks until it finds a `T_LISTEN` code. However, the asynchronous/blocking pulsar model accepts any Open Transport event, and calls `OTListen` once it spies a `T_LISTEN`.

Summary

This paper provided an overview and comparison of the various forms on multitasking on the Macintosh, including a new model. After providing an architectural overview of Open Transport, this paper provided sample code that illustrated how to tie Open Transport to three different multitasking models.