Technotes

Download  Acrobat file (K)	Download  AppleWorks file (74K)

On Improving Open Transport Network Server Performance

Technote 1059 **JULY 1996**

As higher network datalink speeds become more commonplace, Mac OS server performance has come under closer scrutiny. Developers who write network server applications on the Mac continue to be concerned about performance issues. In many cases, server throughput and connection latency problems may stem from poor application design rather than any deficiencies in Open Transport or the Mac OS.

This Technote is intended for Macintosh developers writing network server applications that use the Open Transport API, and discusses some techniques you can employ in your network server application design to achieve higher performance.

CONTENTS

About Open Transport on Mac OS

The Mac OS Open Transport API is based on the industry standard X/Open Transport Interface (XTI) specification. Since XTI originated in the UNIX world, it was slightly altered to operate in an asynchronous environment such as the Mac OS .

When using XTI under UNIX, it is acceptable for a task to issue a blocking I/O request, causing the process to sleep until the request is completed. But since the current Mac OS is based on cooperative rather than preemptive multitasking, blocking I/O is unacceptable. As a consequence, applications must invoke their networking and I/O functions asynchronously; otherwise, the blocking of I/O directly affects the user experience.

Apple's Open Transport addresses this mismatch in software architecture by extending XTI, so that an application can install a notifier to receive completion events.

The key to building an application that takes full advantage of Open Transport on the Macintosh, such as a high performance network server, is the proper use of use of these XTI extensions. Understanding these

extensions is important to both developers who are writing new Macintosh network server applications and those who attempt to port existing code based on UNIX sockets.

Getting the Most Out of Open Transport By Using Notifiers

The Mac OS implementation of XTI allows your application to specify a callback routine or notifier upon opening an endpoint. Since the notifier mechanism is the most immediate way for an application to discover what endpoint events are occurring, your code should attempt to respond to most actions directly in the notifier.

To get the maximum performance out of Open Transport, you ought to take advantage of notifiers. The following sections explain how to work with notifiers.

Don't put off to event time what you can handle in your notifier

In the Mac OS, you have three execution contexts:

- System Task -- WaitNextEvent
- Deferred Task -- Secondary Interrupt, when the interrupt mask is zero
- Primary Interrupt -- I/O completion, VBL or Time Manager tasks

In general, network server code should avoid deferring incoming packet processing to System Task time. The reason is that calling WaitNextEvent can result in an unpredictable packet processing latency. As a result, the round trip time from when your server receives a packet to when it responds depends on factors external to your application. This is a consequence of cooperative multitasking used by the Mac OS. *What this means is that you have no control of how long it takes for another task to call* WaitNextEvent. Open Transport 1.1.1 introduced the SyncIdleEvents feature, which was intended to facilitate Notifier/Thread Manager interaction. What the feature does is call your notifier at a time when it is safe to call YieldtoThread. Since YieldtoThread will eventually cause the Thread Manager to switch to a thread that calls WaitNextEvent, this presents the same unpredictable latency. For this reason, I would suggest not to use this strategy in a high performance server.

A better strategy to processing incoming packets is to receive and initiate all I/O from the notifier. As a rule, if you can start an OTSnd or an asynchronous File Manager operation from a notifier, you should do it. This is the most important piece of advice you can follow if you want to extract the best performance from Open Transport. Network applications developers, especially those designing HTTP servers, should heed this recommendation. Packet-response time is a true measure of Web server performance.

Specifically, in a Notifier you should be able to perform the following tasks:

- Accept and hand off connections
- Receive and process all incoming data
- Start asynchronous I/O operations, e.g., File Manager
- Send network data
- Tear down network connections.

Some Notifier Hints and Guidelines

Because proper use of Open Transport notifiers is the key to improved server performance, here are some hints and guidelines to help you use notifiers more effectively:

- Receive and process network data from the notifier. Don't defer processing to System Task, since this directly affects your turnaround time.
- Treat the notifier code path as a critical section; assume you are locking the operating system from other tasks: keep it short and simple. If you must perform a lengthy operation, consider using a Deferred Task.
- Open Transport will never run a notifier during a Primary Interrupt, only at System Task or Deferred Task time. A Deferred Task, also known as Secondary Interrupt, occurs when the interrupt mask is zero, i.e., on the way out of a Primary Interrupt.
- It is also possible for Open Transport to run a notifier during the execution of or returning from an OT routine. Therefore, you should never call OT at Primary Interrupt time, except to schedule a

Deferred Task.

- You should never make a synchronous OT call from inside a notifier. Doing this will cause Open Transport to return kOTStateChangeErr in order to prevent you from deadlocking.
- You can use completion events to gate endpoint action, i.e., you can use T_OPENCOMPLETE to initiate an OTBind or T_DISCONNECTCOMPLETE to gate an OTUnBind. This method will prevent you from receiving a kOTStateChangeErr for calling a function before an endpoint is ready.
- The normal XTI events (T_DATA, T_LISTEN ...) and the completion events (T_OPENCOMPLETE, T_BINDCOMPLETE ...) will not reenter the notifier. The events that do are T_MEMORYRELEASED and some of the high priority notifications, such as kOTProviderWillClose. For example:
    1. Your notifier receives a T_GODATA.
    2. The notifier responds by calling OTSnd.
    3. Notifier reenters with T_MEMORYRELEASED.
    4. OTSnd returns.
- Starting with Open Transport 1.1.1, you can use the OTEnterNotifier and OTLeaveNotifier functions to prevent your notifier from being entered during a critical section of code.
- For a list of OT functions and in what context you can call them, refer to Appendix F in the *Open Transport Client Developer Note* .

Interrupt-Safe Functions

One of the major reasons that developers have shied away from processing packets in a notifier is you can't call the Mac Toolbox functions that move memory at interrupt time. A number of fast, interrupt-safe functions, however, are available from Open Transport. These functions are the same for both Mac OS 7 and Mac OS 8.

Many developers may overlook the functions available in the OT library. The *Open Transport Client Developer Note* and the <Opentransport.h> include file ought to provide you with the information you need.

Some interrupt-safe functions available under Open Transport are explained in the next section.

Memory Management

OTAllocMem/OTFreeMem can be safely called from a notifier. Keep in mind that the memory pool used by OTAllocMem is allocated from the application memory pool, which, due to the Memory Manager's constraints, can only be filled at task time.

Therefore, if you allocate memory from the Open Transport memory pool from an interrupt or deferred task, you should be prepared to handle a failure resulting from an temporarily depleted memory pool, which can only be replenished at the System Task time.

List Management

OTLIFO functions can be used to implement an interrupt-safe LIFO or FIFO list. Here's how:

1. Create a OTLIFO list.
2. Populate it at interrupt or notifier time by using OTLIFOEnqueue.
3. Use the OTLIFOStealList to atomically remove the list and then call OTReverseList to flip it around, so that it will become a FIFO list.
4. You can then use OTLIFODequeue to remove individual elements from the list.

Semaphores

There are also a number of Atomic functions and macros, such as OTAtomicSet/Clear/TestBit, OTCompareAndSwap, OTAtomicAdd, OTClearLock/OTAcquireLock that can be used to administrate interrupt-safe semaphores.

TimeStamp

There are a number of TimeStamp functions that can be used to accumulate profiling information -- for example, OTGetTimeStamp. These functions are fast and safe to call at Primary Interrupt time. You can use this time stamp information later to identify bottlenecks or report on application performance.

Open Transport Deferred Tasks: A Closer Look

Open Transport Deferred Tasks provides a way to simplify working with primary interrupts, such as IO completions, VBL tasks, or Time Manager tasks. Remember that a Deferred Task, also known as Secondary Interrupt, occurs on the way out of processing a primary interrupt, after the interrupt mask has been lowered to zero. Deferred Tasks are in effect a priority above SystemTask, but still can be interrupted by a Primary Interrupt such as packet reception.

OTCreateDeferredTask can be used to setup a block of code that can be scheduled from primary interrupts to run at next Deferred Task time. Just pass OTCreateDeferredTask a pointer to the function you wish to schedule and an contextInfo argument, and it returns you a reference that can be used later to schedule the function.

```
dtRef = OTCreateDeferredTask(taskproc,  contextInfo);
```

You can then use OTScheduleDeferredTask to schedule the function associated with the reference to run at the next Deferred Task time. Once scheduled, the function pointed to by taskproc will be called back at the appropriate time and passed contextInfo as a parameter.

```
if(  OTScheduleDeferredTask(dtRef) ) ... ;
```

The OTScheduleDeferredTask will return true if the function was scheduled, false if not. If the function was not scheduled, and the dtCookie parameter is valid, then this indicates that the function is already scheduled to run.

Note:

Although you can call OTScheduleDeferredTask at any time a during primary interrupt, you must bracket the call with a OTEnterInterrupt / OTLeaveInterrupt pair, or better yet, use OTScheduleInterruptTask.

Warning:

Since Open Transport does not keep track of outstanding Deferred Task requests, it is your application's responsibility before quitting to ensure that all outstanding Deferred Task requests have either fired, or have been cancelled with the OTDestroyDeferredTask call.

Avoiding Synchronization Problems

If you mix the processing of OTRcv in different interrupt contexts, such as notifier and Deferred Task or SystemTask time, you should be aware that certain synchronization problems can occur.

For example:

1. You call OTRcv from your main thread.
2. There is no pending data, so OT returns a kOTNoDataErr.
3. Just then, an inbound data packet interrupts OT, causing it to step down to deferred task time to process the data.
4. OT calls your notifier with a T_DATA event, which you ignore.
5. Although the OTRcv in your main thread completes with a kOTNoDataErr, you have no way of knowing that you got the T_DATA event, and you won't get another one until you read to kOTNoDataErr again.
6. The result: your application hangs.

Using Open Transport with the File Manager

Since the design of many network server applications requires interaction with the File Manager, it's important

to understand how to get the most out of the File Manager.

Not Blaming the File Manager for Poor Application Design

Improper use of the File Manager will adversely affect network server performance. If the architecture of your server requires even moderate amounts of file access, you should review Technote *FL16 - File Manager Performance and Caching* . This Note details tactics you can apply in order to get the best performance from the File Manager. Pay close attention to the following issues:

- Optimizing the size of your I/O requests
- Aligning your I/O requests
- Using the File System's Cache to your advantage
- Using asynchronous read or writes that overlap with other non-File Manager operations.

Caching Open Files, Not Just Data

An excellent way to improve performance is to avoid opening frequently used files every time they are accessed. You can accomplish this by maintaining the most recent or commonly used files in an open state, tracked by a list or cache, and only closing files after the list is full or an extended period of inactivity has elapsed.

For example, a webserver would benefit by keeping the site's main **index.html** file open because it is hit everytime a new user accesses the server.

There are some tradeoffs, however, to keeping a large number of files open:

- HFS limits you to a fixed number of open files per disk.
- Too many open files may cause other applications to fail.
- Open files may be tricky to share because of synchronization issues.

An Example of Processing a Packet

So far, I've provided you with a collection of hints and guidelines. Now I want to show you how you might use notifiers effectively to handle packet reception and processing.

We're going to set up a example scenario of processing a packet in the context of an HTTP server. Let's make the following assumptions:

- You have open and bound your session endpoint.
- AckSends are enabled.
- You have also created a deferred task using OTCreateDeferredTask to handle network replies to be used later from File Manager ioCompletion proc.

Here's the scenario:

A HTTP GET-method packet is sent to your port.

1. Your notifier receives a T_DATA event.
2. Call into OTRcv to extract data from the stream head, copying the data into a buffer you allocated with the OTAllocMem function.
3. Return to step 2 until the OTRcv returns a kOTNoDataErr.
4. After parsing your data, you determined that it was a GET-request and a file access is required to respond.
5. Your parser calls OTFreeMem to return the request buffer.
6. Your code calls PBHOpenDFAsync with a pointer to a ioCompletion routine chains to the rest of the required I/O.
7. The notifier returns and the main thread continues.
8. Some time later, the main task is interrupted to process the PBHOpenDFAsync completion routine, which allocates a block of memory via OTAllocMem for the HTTP response. PBReadAsync is used

to access the data.
9. The PBReadAsync completes and its ioCompletion routine runs.
10. Since File Manager ioCompletion routines can sometimes be run at primary interrupt time, it is not safe to directly call Open Transport here. Instead, you should queue your network reply data to your sending task and schedule it to be run with OTScheduleInterruptTask. Then return from the ioCompletion routine.
11. The previously scheduled Deferred Task is run, which can now safely call OTSnd to transmit the HTTP GET-response.
12. Close the file with PBCloseAsync.
13. The OTSnd completes and the notifier is called with a T_MEMORYRELEASED event.
14. The Notifier calls OTFreeMem.

Streamlining Endpoint Creation

The time required to create and open an endpoint can directly impact connection set-up time. This is of prime concern for servers, especially HTTP servers, since they must manage high connection turnover rates. Here are three tips that will speed up endpoint creation.

Preallocating Endpoints

One cost of the transport independence provided by Open Transport is that the process of setting up STREAMS plumbing for each endpoint can be time -consuming.

Rather than incur this delay each time a connection is established, a server designed to handle multiple outstanding connections can preallocate a pool of open, unbound endpoints into an endpoint cache. When a connection is requested, you can quickly dequeue a ready-to-use endpoint from this cache, resulting in a decreased connection turnaround delay (e.g., 10 to 30 times faster). For example:

1. Open a number of endpoints with OTAsyncOpenEndpoint.
2. When the notifier receives the T_OPENCOMPLETE event, queue the returned EndpointRef to endpoint-cache.
3. When your accepting notifier receives the T_LISTEN event, you can dequeue an endpoint from the endpoint-cache and pass it to OTAccept. Thus, the only time you have to wait for a endpoint to be created is if the queue is empty, where you can allocate a block of endpoints.

Recycling Endpoints

You can use the endpoint-cache to recycle endpoints when your connection is closed. Rather than call OTCloseProvider each time a connection terminates, cache the unbound endpoint. This recycles it for a later open request.

1. On receiving the T_DISCONNECT, unbind the endpoint with OTUnbind.
2. When your session endpoint receives the T_UNBINDCOMPLETE event, enqueue that endpoint into your endpoint-cache.

Optionally, to save memory, you can deallocate the endpoint when the endpoint-cache reaches some predetermined limit.

Cloning Configurations

Another tactic to consider is to create a single OTConfiguration with OTCreateConfiguration, then use OTCloneConfiguration to pass the OTConfiguration to the OTOpenEndpoint. You'll find that OTCloneConfiguration is approximately 5 times faster than OTCreateConfiguration.

Don't forget to free up the initial OTConfiguration before you quit your application.

Managing the Connection Queue

A high-performance server must be able to handle multiple connections simultaneously -- and efficiently. How you manage the connection queue is a key component of this. The connection queue determines the number of outstanding connect indications or *calls* for a server's listening endpoints -- i.e., connections that have neither been accepted via OTAccept nor rejected via OTSndDisconnect.

Under the XTI framework, managing the connection queue can be complicated. The following section discusses the problems that you might encounter -- as well as the solutions.

Handling kOTLookErr

One consequence of setting up a connection-oriented, listening endpoint to handle multiple outstanding calls is being able to handle the (dreaded) Look Error, or kOTLookErr. The kOTLookErr occurs when another concurrent event has arrived on the same endpoint, and cannot be acted on until the blocking event is consumed.

The Problem

To help illustrate why kOTLookErr occurs, you may want to think of the listening endpoint stream head as an event FIFO. When you bind the listening endpoint, you specify the queue length of this FIFO. If you specify a queue length of greater than one, then multiple T_LISTEN or T_DISCONNECT events will queue up.

Certain rules apply when processing this queue. The rules which regulate your interaction with this queue are codified by the X/Open XTI specification. The following are the relevant rules you need to know in order to write your connection management code (in no particular order):

- A T_LISTEN event is cleared by OTListen.
- A T_DISCONNECT event is cleared by OTRcvDisconnect.
- A *call* is cleared by OTAccept, OTSndDisconnect, OTRcvDisconnect.
- An OTListen can get a kOTLookErr because a T_DISCONNECT is on the top of the stream.
- An OTAccept can get a kOTLookErr because a T_DISCONNECT or T_LISTEN is on the top of the stream.
- An OTRcvDisconnect can get a kOTLookErr because a T_DISCONNECT is on the top of the stream.
- An OTSndDisconnect cannot get a kOTLookErr.

An Example

Properly handling these connect requests requires you to be able to simultaneously process the number of *calls* that you specified in the queue length.

For example:

1. Assume you specified a qlen of 5 in the OTBind.
2. The client end sends you a connection request.
3. Your notifier receives a T_LISTEN event.
4. You invoke OTListen to get the *call.*
5. You do an OTAccept on the *call.*
6. OTAccept returns an kOTLookErr, because another T_LISTEN event is pending.

This can continue until you have filled the queue with 5 inbound connection requests, at which point Open Transport will automatically refuse any further connection requests on that endpoint.

The Solution

You can apply the following strategy to handle processing inbound connection requests.

We're going to set up a example scenario of accepting connections from our listening endpoint. Let's make the following assumptions:

- You are using endpoints in asynchronous/blocking mode.

- You have a queue length greater than 1.
- You are handing connections off from a listening endpoint to acceptor endpoints.
- You do most of the work inside your notifier routine.

Here's the scenario:

The first thing you want to do is create a list (LIST) or array of call (CALL) instances large enough to handle your queue.

1. When your notifier gets a T_LISTEN event, call OTListen in a loop until you get a kOTNoDataErr or a kOTLookErr. For each OTListen which returns kOTNoError, queue the CALL to the LIST for later processing. If the result is kOTNoDataErr, you have already gotten all of the CALLs, so go to step #3. If the result is kOTLookErr, it is because of a T_DISCONNECT event. Go to step #2.
2. Do an OTRcvDisconnect and get the associated CALL's sequence number. Find that CALL in the LIST and delete it. Return to step #1.
3. If the LIST is not empty, take the first CALL and determine if you want to accept it. If you want to accept it, go to step #4. To reject it, go to step #5.
4. Call OTAccept on the CALL. If the result is kOTNoError, delete the CALL from the LIST and return out of the notifier. If the result is kOTLookErr, do a OTLook. If the look result is T_LISTEN, go to step #1. If the look result is T_DISCONNECT, go to step #2.
5. Call OTSndDisconnect on the CALL. If the result is kOTNoError, delete the CALL from the LIST and return out of the notifier. If the result is kOTLookErr, go to step #2, T_DISCONNECT is the only reason OTSndDisconnect can return a look error.
6. When your notifier gets a T_PASSCON, check to see if the T_ACCEPTCOMPLETE has already happened. If not, return out of the notifier. If so, go back to step #3.
7. When your notifier gets a T_ACCEPTCOMPLETE, check the result. If the result is anything other than kOTNoError, you won't be getting a T_PASSCON event. Do the appropriate error handling, then return to step #3. If the result is kOTNoError, check to see if the T_PASSCON has already happened. If not, return out of the notifier. If so, go back to step #3. When your notifier gets a T_DISCONNECTCOMPLETE, go back to step #3.

Alternatively, you could make step #3 a loop and handle all of the CALLs in the LIST simultaneously, but that makes handling the T_ACCEPTCOMPLETE / T_PASSCON events and LIST processing more complicated. In general, the added complexity may not be worth it.

If you don't handle everything inside your notifier, there's one gotcha: if you set a flag in your notifier and process the event back in your main thread, you must deal with the following type of synchronization issue:

There is a T_LISTEN on top of the queue hiding a T_DISCONNECT. Your notifier gets a T_LISTEN and you set a flag to handle it back in the main thread. The main thread does an OTListen, which clears the T_LISTEN and brings the T_DISCONNECT up to the top. Before the OTListen completes, your notifier will be interrupted with the T_DISCONNECT notification.

Once you understand how things work, handling kOTLookErr is not as perplexing as it might seem.

Negotiate qlen on Bind

As mentioned at the beginning of this section, you specify the length of a listening endpoint's connection queue when you bind a connection-oriented service, such as TCP or ADSP. During the bind process, however, it is possible for the value of queue length to be negotiated by the endpoint. This length may be changed if the endpoint cannot support the requested number of outstanding connection indications. Therefore, it is important for your application not to assume that the value of qlen returned by the OTBind is the same as requested. You should always check it.

Increasing Throughput

This section discusses methods for increasing network data throughput to your server to enhance performance.

No-Copy Receive

When properly used, no-copy receives may provide a significant performance enhancement by letting you directly access the network interface's actual DMA buffers. This process lets you determine where and how to copy data; typically, this results in decreasing memory copies by a factor of 2.

To request that Open Transport initiate a no-copy receive, you must pass the constant kOTNetbufDataIsOTBufferStar in the length parameter of the OTRcv. This will return you a pointer to the OTBuffer structure, as illustrated here:

```
OTBuffer* myBuffer;
OTResult result = OTRcv(myEndpoint, &myBuffer,
kOTNetbufDataIsOTBufferStar);
```

The OTBuffer data structure is based on the STREAMS mblk_t data structure, as illustrated below. By tracing the chain of fNext pointers, all of the data associated with the message can be accessed.

```
struct OTBuffer
{
        void*           fLink;          // b_next & b_prev
        void*           fLink2;
        OTBuffer*       fNext;          // b_cont
        UInt8*          fData;          // b_rptr
        size_t          fLen;           // b_wptr
        void*           fSave;          // b_datap
        UInt8           fBand;          // b_band
        UInt8           fType;          // b_pad1
        UInt8           fPad1;
        UInt8           fFlags;         // b_flag
};
```

A few utilities are available to the Open Transport programmer to simplify access to the OTBuffer structure. You can use OTReadBuffer to read data from the buffer, to OTBufferDataSize calculate its total length and OTReleaseBuffer to dispose of the OTBuffer when you are done.

When processing a no-copy receive, it is very important that you minimize the time that you hold onto the buffer and be sure to call OTReleaseBuffer to return it to Open Transport. Otherwise, you run the risk of starving the network driver for DMA buffers and adversely affecting the performance of the operating system.

Warning:

Under no circumstances should you write to the OTBuffer data structure. Doing so will run the result in either an an access fault or system crash .

The *Open Transport Client Developer Note* further documents the no-copy receive method of processing network data.

A Code Snippet Illustrating a No-Copy Receive

This snippet demonstrates how you might process a no-copy receive in your notifier:

```
#include
#include

//  QUEUE_NET_EVENT(T_DATA) defers a particular event to later
//  MyProcessBuffer(buffer,actCount) consumes network data

void HandleDataAvail(EndpointRef ep) {
                OTFlags                         flags;
                OTResult                        status;
                OTBuffer*               buffP;
                OTBufferInfo            buffInfo;
                size_t                  actCount;
                char*                   buffer;
```

```
// Do while data left in OT
        while((status = ::OTRcv(ep, &buffP,
                                            kOTNetbufDataIsOTBufferStar, &flags)) > 0)
         {
        // Get count of bytes in buffer
                OTInitBufferInfo(&buffInfo, buffP);
                actCount = status;

        // Reserve buffer space
        // You should handle OTAllocMem failures. Maybe use a deferred task.
                ThrowIfNil(buffer = OTAllocMem(actCount));

        // Read data into buffer
                ::OTReadBuffer(&buffInfo, buffer, &actCount);

        // Call code to consume network data
                MyProcessBuffer(buffer,actCount);

        // Return OTBuffer to system
                ::OTReleaseBuffer(buffP);
        }

// Did we read all the data ?
                if(status == kOTNoDataErr) return;

// Was the endpoint not ready yet?
                else if(status == kOTStateChangeErr) QUEUE_NET_EVENT(T_DATA);

// Check for Rcv Error
                else ThrowIfErr( status );
};
```

Using AckSend

By enabling the AckSend option on endpoint, you can eliminate the need for Open Transport to perform a memory copy of contiguous data to be transmitted by OTSnd. Instead, a pointer to the data will be passed downstream. Once the memory is no longer being used, the notifier receives a T_MEMORYRELEASED event.

Note:

You don't have to wait for the T_MEMORYRELEASED event before calling OTSnd again, however you can't reuse any memory that you passed OTSnd until Open Transport has released it through the T_MEMORYRELEASED event.

The following snippet provides an example of usage:

```
OTAckSends(ep)                          // enable AckSend option
......
buf = OTAllocMem( nbytes);              // Allocate nbytes of memory from OT
OTSnd(ep, buf, nbytes, 0);              // send a packet
......
NotifyProc( .... void* theParam)        // Notifier Proc
case T_MEMORYRELEASED:          // process event
                OTFreeMem( theParam);   // free up memory
                break;
```

Note:

Because of the complexity of handling endpoint flow-control, Open Transport performance will suffer when the AckSend option is used to handle non-contiguous data, i.e., OTSnd that have been passed a OTData structure. Therefore, use AckSend only when sending contiguous data.

Handling Dead Clients

A properly-designed server should be prepared to handle what happens when a remote client unexpectedly disappears. The problem of a disappearing (or crashed) client is further aggravated when the link has been flow-controlled. This is illustrated in the following scenario :

1. You are transmitting a large amount of data to a client.
2. Your transport provider enters a flow-control state.
3. The client crashes or becomes unreachable.
4. After a timeout your server decides to force a disconnect from that client and issues a disconnect request.
5. But under XTI the stream T_DISCON_REQ is a M_PROTO message and is thus also subject to flow control; this causes your link to hang.

Under XTI it is the application's responsibility to flush the stream before issuing a disconnect. The best way to force a flush is by sending the IFLUSH command to the stream head with an OTIoctl. As illustrated here:

```
#include
error = OTIoctl(ep, IFLUSH,  (void*) FLUSHRW);
if (error)  OTUnBind(ep)
```

Sending Large Blocks of Data

Another way to improve the performance of a high volume network server is by minimizing the overhead incurred by OTSnd. You can do this by sending as large a block as the endpoint's transport service data unit (TSDU) will allow, thus reducing the number of times OTSnd is called.

You can determine the endpoints TSDU size by inspecting the TEndpointInfo returned by OTOpenEndpoint or OTGetEndpointInfo.

For endpoints that support infinite data unit size (T_INFINITE) such as TCP/IP, empirical evidence suggests that a size around 8K bytes might be optimal.

There are a number of tradeoffs involved in selecting the proper size block to pass to OTSnd, but the overall guideline is to do whatever you can to keep the outbound pipe full.

Flow Control

To take optimal advantage of Open Transport's throughput, your server should attempt to keep the outbound data stream full. In order to do this, your application must properly handle flow-control restrictions. Flow-control restrictions are imposed by Open Transport when only a part of the data passed to an endpoint can be accepted by the transport provider. This may be due to many things, including local memory restrictions or even network throughput limitations. In order to prevent data loss, Open Transport will impose a flow-control state on that endpoint.

How Open Transport handles flow control for a given endpoint depends on that endpoint's blocking mode. When an endpoint is in blocking mode, a send request such as OTSnd will wait for flow control to lift, then complete the send.

Conversely, when an endpoint is in non-blocking mode, OTSnd will return immediately with a value that is less than the value of the number of bytes passed to it, or if no bytes at all were sent, it will return kOTFlowErr.

Once in a flow-control state, the endpoint will remain there until the remote side has accepted the pending data. Once the flow-control restrictions are lifted your notifier will be issued a T_GODATA or T_GOEXDATA event.

Under a low memory condition it is possible for OTSnd to return a kENOMEMErr.In this case, you should back off and attempt to send later, possibly by means of a timer. Unlike the kOTFlowErr case, your application is not flow controlled and thus your notifier will not get a T_GODATA event.

Note:

If your application is calls OTSnd outside of your notifier -- for instance, at System Task time -- it should be prepared to handle problems similar to the OTRcv/T_DATA case outlined in the section Avoiding Synchronization Problems.

Other Tips to Improve Performance

There are a few other points that you should be aware of to get the most out of Open Transport in your network server design.

Using the Native OT API Rather Than MacTCP

There is no excuse for your server application not to use the native OT API. Here are some of the reasons why:

- MacTCP is based on the 68K Device Manager model, which on a PPC requires extensive mode switches.
- MacTCP has an inherent limitation of 64 concurrent endpoints.
- MacTCP cannot be extended to support multihomming.
- Open Transport provides a modern implementation of the TCP stack.

But for servers, the primary reason to use the OT API is performance. For example: Experiments over Ethernet on 68K systems have yielded a throughput of somewhere between 300-400k bits/sec using MacTCP vs 600-800k bits/sec using Open Transport. On the PPC, the numbers are more impressive, exceeding the Ethernet speed of 1.5M bits/sec. Preliminary tests indicate speeds approaching 90% link utilization ATM.

Tuning Server Memory Allocation

Another option for improving server performance is to modify Open Transport's memory allocation limits. If you intend for your server application to run on a dedicated Macintosh, this is an effective way to ensure that enough buffer space is available to maintain a high transaction rate.

Starting with version 1.1.1, you will be able to use OTSetMemoryLimits to direct Open Transport to grow its internal client memory pool. You should use this API instead of the OTSetServerMode. OTSetMemoryLimits takes two arguments, a size to grow now and a maximum limit to expand the memory pool.

Remember to call OTSetMemoryLimits(0,0) before quitting your server to inform Open Transport that specialized memory requirements are no longer required.

The prototype for OTSetMemoryLimits is not currently available in the "*Open Transport.h* " include file. You should use the following:

```
extern  OSStatus OTSetMemoryLimits(size_t growSize, size_t maxSize);
```

Warning:
Although currently available, you should be aware OTSetServerMode and OTSetMemoryLimits may not be supported the future versions of Open Transport. To be sure, check with Apple Developer Technical Support.

Server Shutdown

Server developers typically neglect to handle issues related to quitting their applications. To shut down an Open Transport network server properly, you need to be sure that your application performs the following items:

- Ensure that all network and I/O has either completed or aborted.
- Flush any flow-controlled data streams with the IFLUSH ioctl.
- Unbind and Close all endpoints.
- Cancel any Deferred Tasks with OTDestroyDeferredTask.
- Release any OTBuffer structures with OTReleaseBuffer.
- Dispose of any unused OTConfiguration structures with OTDestroyConfiguration.
- Call OTSetMemoryLimits(0,0) to reset memory allocations.

Protocol Specific Issues

Depending on the provider and the version of Open Transport that you are using there are specific optimization and techniques that you should be aware of that be applied to enhance server performance. The best source for this information is the *Open Transport Client Developer Note* and the *Open Transport Release Notes.* You should also check the Apple Open Transport Website for the latest information.

AppleTalk and ADSP

All the topics addressed in this Technote also apply to ADSP.

TCP/IP - Avoiding 2 minute delay on bind

In order to prevent stale data from corrupting a new connection, TCP imposes a 2-minute timeout on a binding after a connection has closed, before allowing the same port to bound to again. This can be worked around by setting the IP_REUSEADDR option with an OTOptionManagement call.

This technique is further documented in the Macintosh Technical Q&A, NW 28 - TCP Application Acquires Different Port Address After Relaunch.

Summary

Because performance issues are a source of concern for developers who write network server applications, the techniques outlined in this Note offer you a set of suggestions you ought to consider in order to improve server throughput. Using notifiers properly is essential to packet processing latency. In addition, using the File Manager efficiently in conjunction with Open Transport will directly increase your throughput.

The more you know about how Open Transport interacts with the Mac OS, the better you'll be able to design applications that take full advantage of its performance.

Further References

- Apple Open Transport Website <http://developer.apple.com/dev/opentransport/>
- Open Transport Client Developer Note
- "Open Transport.h " header file
- Inside Macintosh: Open Transport
- Inside Macintosh: Files
- X/Open CAE Specification, X/Open Transport Interface (XTI), Version 2./Open Company Ltd ISBN# 0-13-353459-6, <http://www.opengroup.org/index.htm/>
- *Technote FL 16 - File Manager Performance and Caching*
