Technote 1145

Living in a Dynamic TCP/IP Environment

By Quinn "The Eskimo!"
Apple Worldwide Developer Technical Support

CONTENTS

This Technote describes some of the intricacies of dealing with TCP/IP in a dynamic environment, such as that provided by Open Transport. Specifically, it describes how to write Open Transport code which correctly handles multiple IP addresses, dial-up links, sleep and wakeup on PowerBooks, modem disconnection, and user reconfiguration.

This Note is directed at all developers using Open Transport TCP/IP services.

Dynamic TCP/IP Fundamentals

Many TCP/IP programs make false assumptions about the TCP/IP environment. While assumptions like "this machine has TCP/IP, therefore it must have an IP address" and "the IP address won't change" were valid for workstations on a university Ethernet, they do not hold for a PowerBook running Mac OS connected via PPP. Specifically, the following are non-obvious consequences of Mac OS's dynamic TCP/IP environment:

1. The computer does not have an IP address until it has acquired one. For example, if the computer is configured to obtain an address via Dynamic Host Configuration Protocol (DHCP), it does not obtain the address until the TCP/IP stacks loads.
2. The computer's IP address can change over time. For example, if the computer's DHCP lease expires, the TCP/IP will negotiate for a new address, which may not be the same as the old address.
3. The user can configure dial-up links (such as ARA, OT/PPP, and most others) to "Connect automatically when starting TCP/IP applications". If your application creates TCP/IP providers without an explicit user request, the user may be startled to find their modem dialing unexpectedly.
4. If the TCP/IP stack unloads, any TCP/IP providers will be automatically closed. The TCP/IP stack unloads in a variety of circumstances, including when the underlying link shuts down (for example, a PPP disconnect), when the user commits changes in the TCP/IP control panel, and when a PowerBook goes to sleep.

This Technote discusses some high-level approaches you can use to cope with these consequences. Before you start, however, you need to know some important rules about Open Transport's current TCP/IP implementation:

- **The act of opening a TCP/IP provider will load the TCP/IP stack.**
- **Loading the TCP/IP stack may cause the modem to dial.**
- **If TCP/IP is using the modem, the TCP/IP stack will unload when the modem disconnects.**
- **When the TCP/IP stack unloads, it closes all TCP/IP providers.**

IMPORTANT:
This Technote only discusses Open Transport's native TCP/IP interface. It does not discuss MacTCP, or Open Transport's MacTCP interface. The MacTCP programming interface is insufficient to handle all of the cases described in this note. For this and other reasons, DTS strongly recommends that developers adopt the Open Transport programming interface.

Note:
Although this Technote focuses on TCP/IP, some of the recommendations are important for AppleTalk as well. Specifically, AppleTalk programs should Be Prepared for a Close.

IP Address Notes

With Open Transport's support for reconfiguration without reboot and "on demand" address acquisition (for example, DHCP), you must be careful not to assume that the machine has a single static IP address. Specifically:

1. If `OTInetGetInterfaceInfo` returns an error when you pass it an index of 0, the TCP/IP stack is not currently loaded. In this state, the computer **does not have an IP address**. You can force the machine to acquire an IP addresses by loading the TCP/IP stack, but this may have adverse consequences. See No Nuisance Calls, Please.
2. The machine may have **more than one IP address**. A computer running stock Mac OS can currently only have more than one IP address in the "single link multi-homing" case (multiple IP

addresses on the same physical interface), although future versions of Open Transport may support multi-link multi-homing, and you can enable multi-link multi-homing today using third-party software. You can determine the list of IP addresses for the machine using the calls `OTGetInterfaceInfo` and `OTInetGetSecondaryAddresses`, as shown in the code snippet below.

3. The machine's list of IP addresses may change over time. Currently there is no way to detect this change, so you must **avoid caching the list of local IP addresses for extended periods**. Your software should not get the machine's IP address once it's launched and use the same address until it quits. Instead, it must reacquire the address each time it's needed.
4. **The "correct" IP address for the machine may depend on whom you're talking to.** This situation is described in detail in a later section.

Getting a List of All IP Addresses

The following code snippet shows how to determine the list of IP addresses for the machine using the calls `OTGetInterfaceInfo` and `OTInetGetSecondaryAddresses`.

```
enum
{
    kOTIPSingleLinkMultihomingVersion = 0x01300000
};

static OSStatus PrintIPAddresses(void)
{
    OSStatus          err;
    Boolean           haveIPSingleLinkMultihoming;
    NumVersionVariant otVersion;
    Boolean           done;
    SInt32            interfaceIndex;
    InetInterfaceInfo info;

    haveIPSingleLinkMultihoming =
            ( Gestalt(gestaltOpenTptVersions, (long *) &otVersion) == noErr
                && ( otVersion.whole >=
                            kOTIPSingleLinkMultihomingVersion )
                && ( OTInetGetSecondaryAddresses !=
                            (void *) kUnresolvedCFragSymbolAddress));

    err = noErr;
    interfaceIndex = 0;
    do {
        done = ( OTInetGetInterfaceInfo(&info, interfaceIndex) != noErr );
        if ( ! done ) {
            printf("fAddress = %08lx\n", info.fAddress);
            if ( haveIPSingleLinkMultihoming ) {
                err = PrintSecondaryAddresses(&info, interfaceIndex);
            }
            interfaceIndex += 1;
        }
    } while (err == noErr && !done);

    return err;
}
```

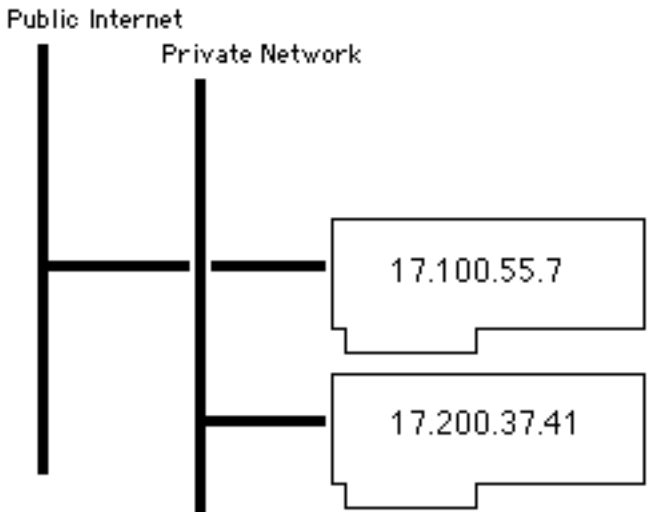
IMPORTANT:
`OTInetGetSecondaryAddresses` is not implemented prior to Open Transport 1.3. To work correctly with systems prior to that, you must weak- link with the `OpenTptInternetLib` and check for its existence by comparing its address to `kUnresolvedCFragSymbolAddress`. For more information about weak-linking, you should read Technote 1083: Weak-Linking to a Code Fragment Manager-based Shared Library.

```
static OSStatus PrintSecondaryAddresses(InetInterfaceInfo* interfaceInfo,
                                        SInt32 interfaceIndex)
{
    OSStatus err;
    InetHost *secondaryAddressBuffer;
    UInt32   numberOfSecondaryAddresses;
    UInt32   addressIndex;

    secondaryAddressBuffer = nil;

    numberOfSecondaryAddresses = interfaceInfo->fIPSecondaryCount;

    if ( err == noErr && numberOfSecondaryAddresses > 0 ) {

        // Allocate a buffer for the secondary address info.

        secondaryAddressBuffer =
                        (InetHost *) OTAllocMem( numberOfSecondaryAddresses
                                                 * sizeof(InetHost) );
        if (secondaryAddressBuffer == nil) {
            err = kENOMEMErr;
        }

        // Ask OT for the list of secondary addresses on this interface.

        if (err == noErr) {
            err = OTInetGetSecondaryAddresses(secondaryAddressBuffer,
                                              &numberOfSecondaryAddresses,
                                              interfaceIndex);
        }

        // Start a server for each secondary address.

        for (addressIndex = 0; addressIndex < numberOfSecondaryAddresses;
                               addressIndex++) {
            printf("secondaryAddressBuffer[%ld] = %08lx\n",
                   addressIndex,
                   secondaryAddressBuffer[addressIndex]);
        }
    }

    // Clean up.

    if (secondaryAddressBuffer != nil) {
        OTFreeMem(secondaryAddressBuffer);
    }

    return err;
}
```

Multi-homing and IP Addresses

On a multi-link multi-homed computer, it's possible for the "correct" IP address of the machine to depend on the IP address of the machine you're communicating with.

For example, consider a machine with two Ethernet cards, one connected to the public Internet and one connected to a private network. Each card has an IP address but the machine does not forward IP packets between the cards. This situation is illustrated below:



Now consider an FTP client running on this computer. One of the (mis)features of the FTP protocol is that, when it opens a **data connection**, the FTP client must open a port on the local machine and send (through the **control connection**) the port number and the local IP address to the server in a PORT command. However, machines on the public Internet can only communicate with 17.100.55.7, and machines on the private network can only communicate with 17.200.37.41. So how does the FTP client work out which IP address to send?

The answer is that the `OTGetProtAddress` call will return the correct local IP address for a connected endpoint. So the FTP client can call `OTGetProtAddress` on the control connection to determine the correct local address to use for the data connection.

The following code snippet uses `OTGetProtAddress` to find the correct local IP address to send based on a connected endpoint.

```
static OSStatus GetLocalIPAddressForConnection(EndpointRef ep,
                                               InetAddress *localAddr)
{
    OSStatus err;
    TBind localBind;

    OTAssert("GetLocalIPAddressForConnection: Only works while connected",
            OTGetEndpointState(ep) == T_DATAXFER);

    OTMemzero(&localBind, sizeof(TBind));
    localBind.addr.buf = (UInt8 *) localAddr;
    localBind.addr.maxlen = sizeof(InetAddress);

    err = OTGetProtAddress(ep, &localBind, nil);
    return err;
}
```

Note:
This routine asserts that the endpoint is in the `T_DATAXFER` state because, if the endpoint is not connected, `OTGetProtAddress` will simply return the address to which the endpoint was bound, which is generally not at all helpful.

IMPORTANT:
Due to limitations in the current Open Transport architecture, the IP address returned by the above may not be in the list of all IP addresses generated by the code in the previous section.

No Nuisance Calls, Please

Your software must be careful to create a TCP/IP provider only when it actually needs to use the network. This is because many users have a dial-up network connection configured to "dial on demand". The act of you opening a TCP/IP provider will cause the TCP/IP stack to load, and hence the modem to dial, even if you don't send any network traffic.

Note:
In OT/PPP and ARA 3.x, the "dial on demand" feature is controlled by a checkbox in the Options dialog labeled "Connect automatically when starting TCP/IP applications".

Dial on demand works best if applications open TCP/IP providers only in direct response to a user operation. For example, if the user requests information from the net, your application is free to trigger a dial because the user is expecting it. In this Technote, these requests are referred to as **solicited operations**.

Dial on demand works badly for applications that access the Internet "in the background." For example, your application may want to periodically update a local database with information from the network. Triggering a dial for such an **unsolicited operations** is going to annoy the user.

Your application can avoid dialing the modem for unsolicited operations by simply calling `OTInetGetInterfaceInfo` before starting the network operation. If `OTInetGetInterfaceInfo` indicates that the TCP/IP stack is loaded, the modem (if any) has already been dialed and your application can safely use the network. The following snippet shows how to use `OTInetGetInterfaceInfo` to determine if the TCP/IP stack is loaded.

```
static Boolean IsTCPStackLoaded(void)
{
    InetInterfaceInfo info;

    return ( OTInetGetInterfaceInfo(&info, 0) == noErr );
}
```

On the other hand, if the TCP/IP stack is not loaded, your application is faced with a dilemma: the computer may be on a non-dial-up link (such as Ethernet) where loading the TCP/IP stack is not a serious inconvenience, or the computer may be on a dial-up link where loading the TCP/IP stack will cause a "nuisance call." In many cases, you can ignore this dilemma entirely and unilaterally decide to avoid unsolicited operations while TCP/IP is unloaded. A majority of desktop users are regularly using the Internet for solicited operations, which keeps the TCP/IP stack loaded, ready to handle unsolicited operations, so this isn't really a problem.

However, some developers have found this heuristic to be insufficiently rigorous for their taste. If you fall into this category, you can take further steps to ensure that your unsolicited operations get to the network. Specifically:

1. You can use the Network Setup library (introduced with Mac OS 8.5) to read the TCP/IP preferences to determine whether TCP/IP is configured to use a dial-up link. The DTS sample code "OTTCPWillDial" shows how to do this.
2. You can time how long an unsolicited operation has been waiting to access the network. If the operation has been waiting too long, you can request user interaction (probably using `AEInteractWithUser`) and ask the user whether they would like you to dial the modem to complete this operation.

Be Prepared for a Close

As part of its normal operation, the TCP/IP stack may decide to close your provider. When it closes a provider in this way, Open Transport calls the provider's notifier with one of two events:

1. `kOTProviderWillClose` is sent when OT is closing your provider in a controlled fashion, typically when the user is reconfiguring the TCP/IP stack. It is always sent at system task time. You may choose to put the provider in synchronous mode and shut down the network connection cleanly.
2. `kOTProviderHasClosed` is sent when OT "force closes" your provider. This is typically done when the underlying link layer shuts down. It may be sent at system task or deferred task time, so you typically write your code to assume it's at deferred task time. The underlying provider has already been closed but you must call `OTCloseProvider` to avoid a (small) memory leak.

Regardless of the action of your notifier, the provider will always be closed when you return from your notifier. Any operations on the provider after your notifier returns will return `kOTBadReferenceErr`.

For TCP/IP providers on which you're actively working (for example, worker endpoints which are actively transferring data), you can ignore these events. The next time you use the provider, you will get a `kOTBadReferenceErr` error, which your generic error handling should respond to by closing the provider. However, if you open a TCP/IP provider which you're not actively working on (such an `InetSvcRef` which you use periodically for DNS lookups, or a listening endpoint in a server), you must install a notifier which handles these events. The following code snippet is an example of how to do this.

```
static InetSvcRef gInetServices = kOTInvalidProviderRef;

static pascal void MyInetServicesNotifier(void* contextPtr, OTEventCode code,
                                          OTResult result, void* cookie)
{
    switch (code) {
        case kOTSyncIdleEvent:
            [... yielding code removed ...]
            break;

        case kOTProviderWillClose:
        case kOTProviderIsClosed:
            // OT is closing the provider out from underneath us.
            // We remove our reference to it so the next time
            // someone calls MyStringToAddress, we'll reopen it.
            (void) OTCloseProvider(gInetServices);
            gInetServices = kOTInvalidProviderRef;
            break;
        default:
            // do nothing
            break;
    }
}

static OSStatus MyOpenInetServices(void)
```

```
{
    OSStatus err;
    OSStatus junk;

    printf("MyOpenInetServices: Opening gInetServices.\n");
    gInetServices = OTOpenInternetServices(kDefaultInternetServicesPath, 0, &err);
    if (err == noErr) {
        (void) OTSetBlocking(gInetServices);
        (void) OTSetSynchronous(gInetServices);
        (void) OTInstallNotifier(gInetServices, MyInetServicesNotifier, nil);
        (void) OTUseSyncIdleEvents(gInetServices, true);
    }
    return err;
}

static OSStatus MyStringToAddress(const char *string, InetHost *address)
{
    OSStatus err;
    InetHostInfo hostInfo;

    // If the DNS provider isn't currently open, open it.

    err = noErr;
    if (gInetServices == kOTInvalidProviderRef) {
        err = MyOpenInetServices();
    }

    // Now do the name to address translation using the provider.

    if (err == noErr) {
        err = OTInetStringToAddress(gInetServices, (char *) string, &hostInfo);
    }
    if (err == noErr) {
        // For this example, we just return the host's first IP address.
        *address = hostInfo.addrs[0];
    }
    return err;
}
```

The sample illustrates two important points:

1. The `gInetServices` provider is not created until the program needs to convert a name to an address. This sort of "lazy" creation prevents the program dialing the modem prematurely.
2. When the notifier is informed that the provider has closed, it invalidates `gInetServices`. The next time `MyStringToAddress` is called, it will notice this and re-create the provider.

Server Etiquette

Handling the events described in the previous section is especially important for servers. A server will typically have one or more listening endpoints (**listeners**), which are waiting for connections from clients. If the TCP/IP stack unloads, those listeners will close, and your notifier will receive an event to this effect. If you don't handle these events properly, chances are that your server will keep running just fine, except it will receive no more `T_LISTEN` events and will be "deaf" to its clients.

The standard response to the closing of a listener is to simply reopen the listener. You must not attempt to do this directly in your notifier. Instead you should set a flag that causes your main event loop to re-open the listener as soon as the TCP/IP has loaded. You can determine this by checking the result of `OTInetGetInterfaceInfo`, as described in No Nuisance Calls, Please.

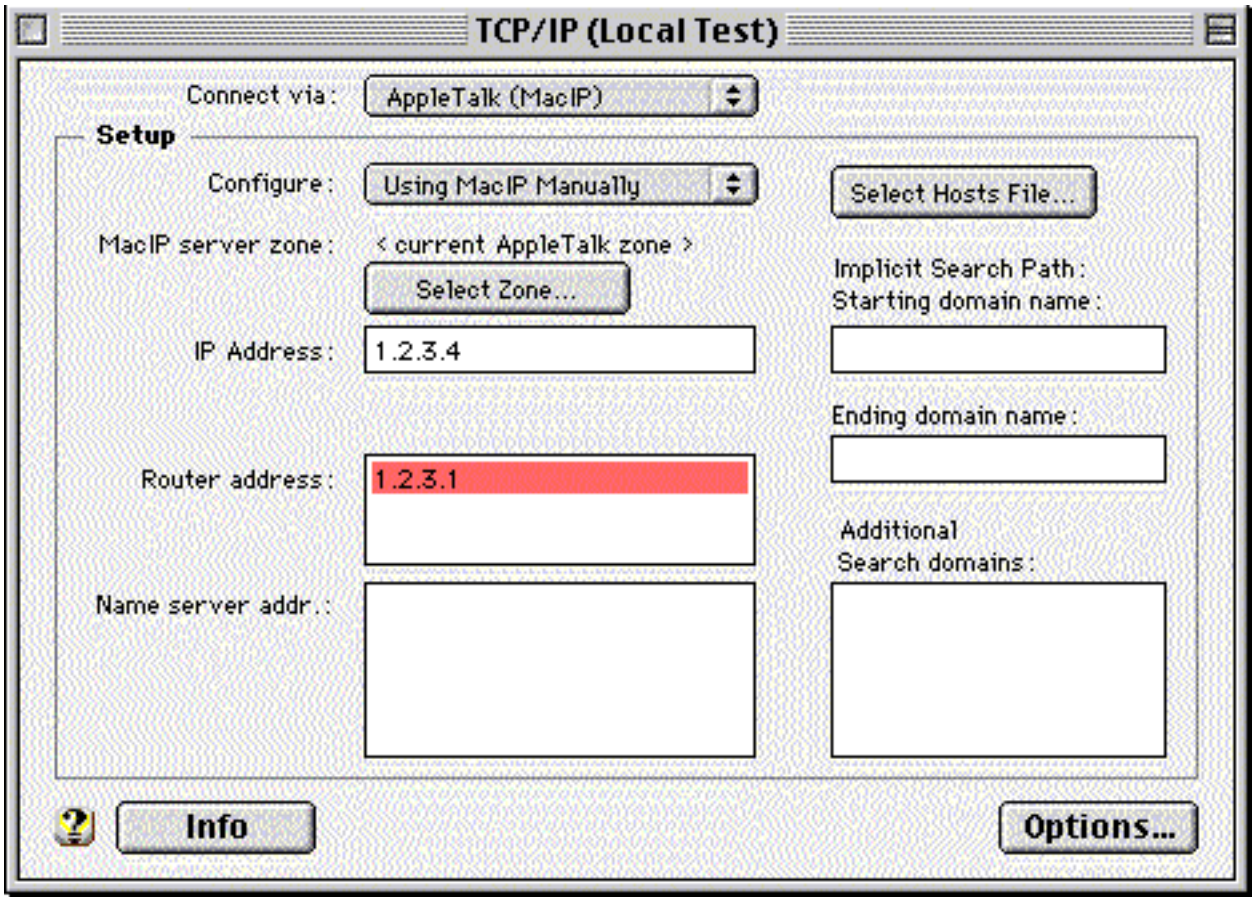
Typically, opening a listener is an unsolicited operation and should be deferred if it would cause the modem to dial. If your application is likely to be used in an environment where it should dial the modem to open a listener, you may want to provide a user preference for this. An example is shown below. Typically the first option would be the default.

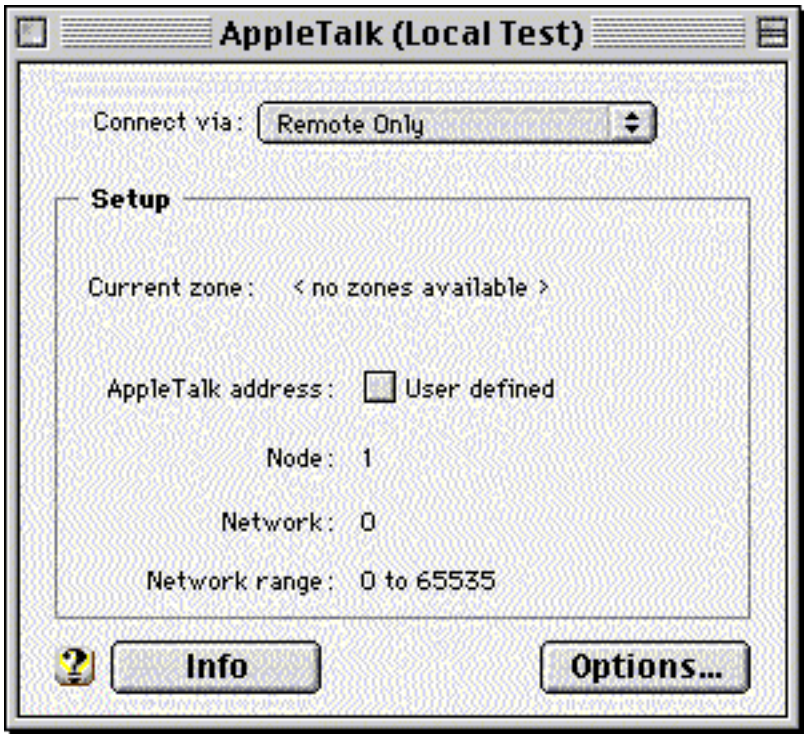
Talking to Yourself

Talking to yourself may be the first sign of madness, but TCP/IP software often wants to talk to other software running on the same machine. For example, you might have a server administration tool that configures your server over the network. It's reasonable for a user to run both the server and the administration tool on the same machine.

OT supports this sort of loopback, including the standard 127.0.0.1 loopback address. However, problems arise when you use this technique on a machine configured for a dial-up connection. Remember that opening a TCP/IP provider causes the TCP/IP stack to load, and loading the TCP/IP stack may cause the modem to dial. This is true even if you're just using the endpoint to talk to yourself.

A future version of Open Transport may alleviate this problem as part of the multi-link multi-homing solution but, for the moment, there is no good workaround. One less-than-ideal workaround is to reconfigure TCP/IP to not connect via the modem. If no other suitable link is available, you can always configure TCP/IP to "MacIP" (with manual addressing) and configure AppleTalk to "Remote Only". The following screen shots show what this might look like.





It is possible to use the Network Setup library to programatically create and switch to these settings.

Summary

TCP/IP is no longer limited to desktop machines on Ethernet. Your code must avoid making false assumptions about the TCP/IP environment, and must adapt to radical changes in the TCP/IP environment as the user reconfigures and relocates their computer. Your code must also strive to avoid annoying users with dial-up connections by dialing their modem unexpectedly. The information in this Technote will help you write software that is a pleasure to use in a dynamic TCP/IP environment.

Further References

- Inside Macintosh: Networking with Open Transport
- Technote 1083: Weak-Linking to a Code Fragment Manager-based Shared Library
- Internet STD 9 File Transfer Protocol (also RFC 959)

Downloadables

 Acrobat version of this Note (34K).

 Binhexed Routine Descriptor Lib (179K).

Acknowledgments

Thanks to Richard Buckle, Stuart Cheshire, Mark Cookson, Rich Kubota, Peter N Lewis, Pete Resnick, and Brad Suinn.

To contact us, please use the Contact Us page.

Updated: 2-November-98

Technotes

Previous Technote | Contents