Inktomi®

Traffic Edge Software Developer's Kit
Programmer's Guide

Release 1.5

June 2002

Content Networking Solutions Group
4100 East Third Avenue
Foster City, CA 94404

Phone: (650) 653-2800
Fax: (650) 653-2801
Web: http://www.inktomi.com

Contents

Preface

This manual is a reference for creating plugins, programs that add services such as filtering or content transformation, or entire features such as new protocol support, to Inktomi Traffic Edge. You create plugins using the Traffic Edge Software Development Kit (SDK) which consists of:

- This manual, the *Traffic Edge SDK Programmer's Guide*
- `InkAPI.h`, the header file containing the Traffic Edge API
- Sample Traffic Edge plugin code
- `SDKtest`, a tool for testing plugins; SDKtest includes synthetic clients and servers
- Header files containing the SDKtest APIs (client and server APIs)
- Sample Traffic Edge `SDKtest_client` and `SDKtest_server` plugins
- The *Traffic Edge SDKtest User's Guide*, the guide to using SDKtest and writing SDKtest plugins

This preface contains the following information:

- *Who should read this book, on page 9* tells you what background you need in order to understand the material in this manual
- *How to use this book, on page 9* outline the strucure of this manual and gives guidelines on how to use it for various purposes (basic learning about plugins, how to write specific kinds of plugins, how to find reference information)
- *Conventions used in this manual, on page 11* lists the typographic conventions used in this manual

Who should read this book

This manual is intended for programmers who want to write plugin programs that add services to Traffic Edge.

This manual assumes a cursory knowledge of the C programming language, the Hyper-Text Transfer Protocol (HTTP), and Multipurpose Internet Mail Extensions (MIME).

How to use this book

This book has four parts:

- Introduction and overview

- Tutorials on writing specific kinds of plugins: HTTP header-based plugins, content transformation plugins, and protocol plugins
- Guides on specific interfaces
- Reference chapter and appendixes

If you are new to writing Traffic Edge plugins, read the first two chapters, *Getting Started* and *Creating Traffic Edge Plugins*, and use the remaining chapters as needed. The third chapter, *Header-Based Plugin Examples*, for details about plugins that work on HTTP headers. Read the fourth chapter, *HTTP Transformation Plugins*, if you want to write a plugin that transforms or scans the body of an HTTP response. Read *"New Protocol Plugins" on page 55* if you want to support your own protocol on Traffic Edge.

Look up information in the following indexes:

- *"Concept Index" on page 275*, listing information by subject
- *"Function Index" on page 281*, listing all Traffic Edge API calls
- *"Constant Index" on page 277*
- *"Type Index" on page 287*

In the PDF and HTML formats of this book, cross references are active links. Click on links to access the cross reference.

Following is a chapter-by-chapter breakdown of chapter contents:

- *"Getting Started" on page 13*

  How to compile and load plugins. Walks through a simple `hello world` example. Explains how to initialize and register plugins.

- *"Creating Traffic Edge Plugins" on page 23*

  Basic structures that all plugins use. Events, continuations, and how to hook on to Traffic Edge processes. Detailed explication of the sample blacklisting plugin.

- *"Header-Based Plugin Examples" on page 31*

  Detailed explication of writing plugins that work on HTTP headers. Discusses the sample blacklisting and basic authorization plugins.

- *"HTTP Transformation Plugins" on page 41*

  Detailed explication of the null-transform example. Discusses vconnections, VIOs, and IO buffers.

- *"New Protocol Plugins" on page 55*

  Detailed explanation of sample protocol plugin that supports a synthetic protocol. Discusses vconnections, mutexes, and the new net connection, DNS lookup, logging, and cache APIs.

The remaining chapters are the API function reference, organized according to function type.

- *"Miscellaneous Interface Guide" on page 79*

  Functions include error writing and tracing functions, thread functions, and Traffic Edge API versions of the `malloc` and `fopen` families. The Traffic Edge API versions overcome various C library limitations (such as portability to all Traffic Edge-supported platforms).

- *"HTTP Hooks and Transactions" on page 65*

  Use the functions in this chapter to hook your plugin to Traffic Edge HTTP processes.

- *"HTTP Headers" on page 83*

  These functions examine and modify HTTP headers, MIME headers, URLs, and the marshal buffers that contain header information. This chapter contains instructions for implementing performance enhancements for all plugins that manipulate HTTP headers. Be sure to read this chapter if you are working with headers.

- *"Mutex Guide" on page 101*
- *"Continuations" on page 109*

  Continuations provide the basic call back mechanism and data abstractions used in Traffic Edge.

- *"Plugin Configurations" on page 115*
- *"Actions Guide" on page 117*

  How to use `INKActions` and the `INKDNSLookup` API.

- *"IO Guide" on page 121*

  How to use the Traffic Edge IO interfaces: `INKVConnection`, `INKVIO`, `INKIOBuffer`, `INKNetVConnection`, the Cache API.

- *"Plugin Management" on page 131*

  These functions allow you to set up a configuration interface for plugins, access installed plugin files, and set up plugin licensing.

- *"Adding Statistics" on page 137*

  Use these functions to add statistics to your plugin.

- *"Function Reference" on page 141*

  A listing of all of the functions in the Traffic Edge API, grouped according to their functionality.

The following two appendixes are provided for reference:

- *"Sample Source Code" on page 245*
- *"Deprecated Functions" on page 253*

Conventions used in this manual

This manual uses the following typographic conventions:

Convention	Purpose
italics	Italics introduce terms.
`monospaced face`	Represents C language statements, commands, file content and computer output.
`monospaced bold`	Represents commands that you should enter literally, as in the example, type **`simplequery`**.

Convention	Purpose
monospaced italic	Represents variables for which you should substitute a value, as in the example, “enter a *filename*.”
ellipsis . . .	Indicates the omission of inconsequential information.

CHAPTER 1

Getting Started

The Inktomi Traffic Edge API lets you create plugins, using the C programming language, that customize the behavior of your Traffic Edge. This chapter contains the following sections:

- *"Understanding Traffic Edge plugins" on page 13*

  This section is a brief introduction to plugins. For more details, see *"Creating Traffic Edge Plugins" on page 23*.

- *"A simple plugin" on page 17*

  This section walks through compiling and loading a `hello world` plugin.

- *"Plugin Registration and Version Checking" on page 20*

  You need to make sure that the Traffic Edge version you are running supports the SDK version for your plugin. This section shows you how to register your plugin's SDK version and have it check the Traffic Edge version.

- *"Naming conventions" on page 21*

For guidelines on creating plugin source code, see *"Creating Traffic Edge Plugins" on page 23*.

Understanding Traffic Edge plugins

Traffic Edge provides sophisticated caching and processing of web-related traffic, such as DNS and HTTP requests and responses.

Traffic Edge itself consists of an event-driven loop that might be simplified as follows:

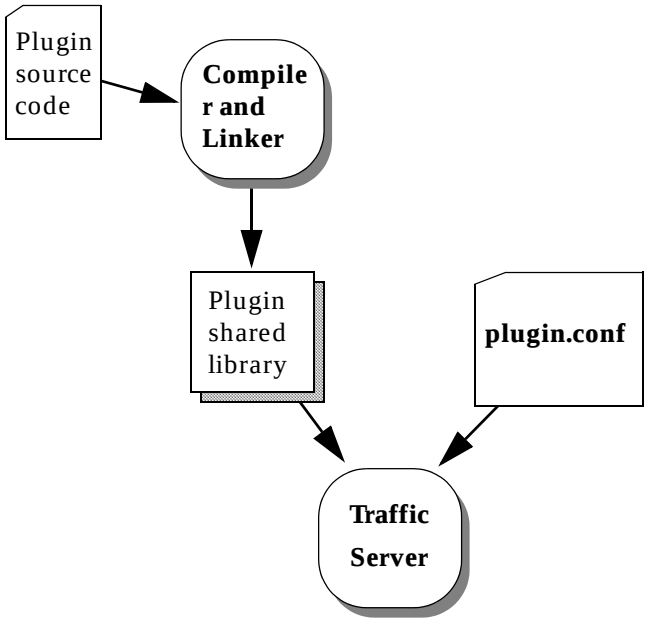
```
for (;;) {
    event = get_next_event();
    handle_event (event);
  }
```

The role of plugins

You compile your plugin source code to create a shared library that Traffic Edge loads when it is started. Your plugin contains callback functions that are registered for particular Traffic Edge events.

When Traffic Edge needs to process an event, it invokes any and all call-back functions you have registered for that event type.

CAUTION Since plugins add object code to Traffic Edge, programming errors in a plugin can have serious implications. Bugs in your plugin, such as an out-of-range pointer, might cause Traffic Edge processes to crash or result in undefined and unpredictable behavior.



Possible uses for plugins

Traffic Edge is a high-performance proxy cache. Plugins are applications built on top of Traffic Edge that extend Traffic Edge's capabilities in:

- HTTP processing (plugins can filter, blacklist, authorize users, transform content)
- Protocol support (plugins can enable Traffic Edge to proxy-cache new protocol content)

Some examples of plugins include:

- A blacklisting plugin, that denies attempts to access web sites that are off-limits.
- An append transform plugin, that adds text to HTTP response content.
- An image conversion plugin, that transforms JPEG images to GIF images.
- A compression plugin, that sends response content to a compression server that compresses the data (alternatively the compression could be done by a compression library local to the Traffic Edge host machine).
- An authorization plugin, that checks user's permissions to access particular web sites. The plugin could consult a local authorization program or send queries to an authorization server.
- A plugin that gathers client information from request headers and enters this information in a database.

- A protocol plugin, that listens for specific protocol requests on a designated port, and uses Traffic Edge's proxy server and cache to serve client requests.

The following figure illustrates various types of plugins:

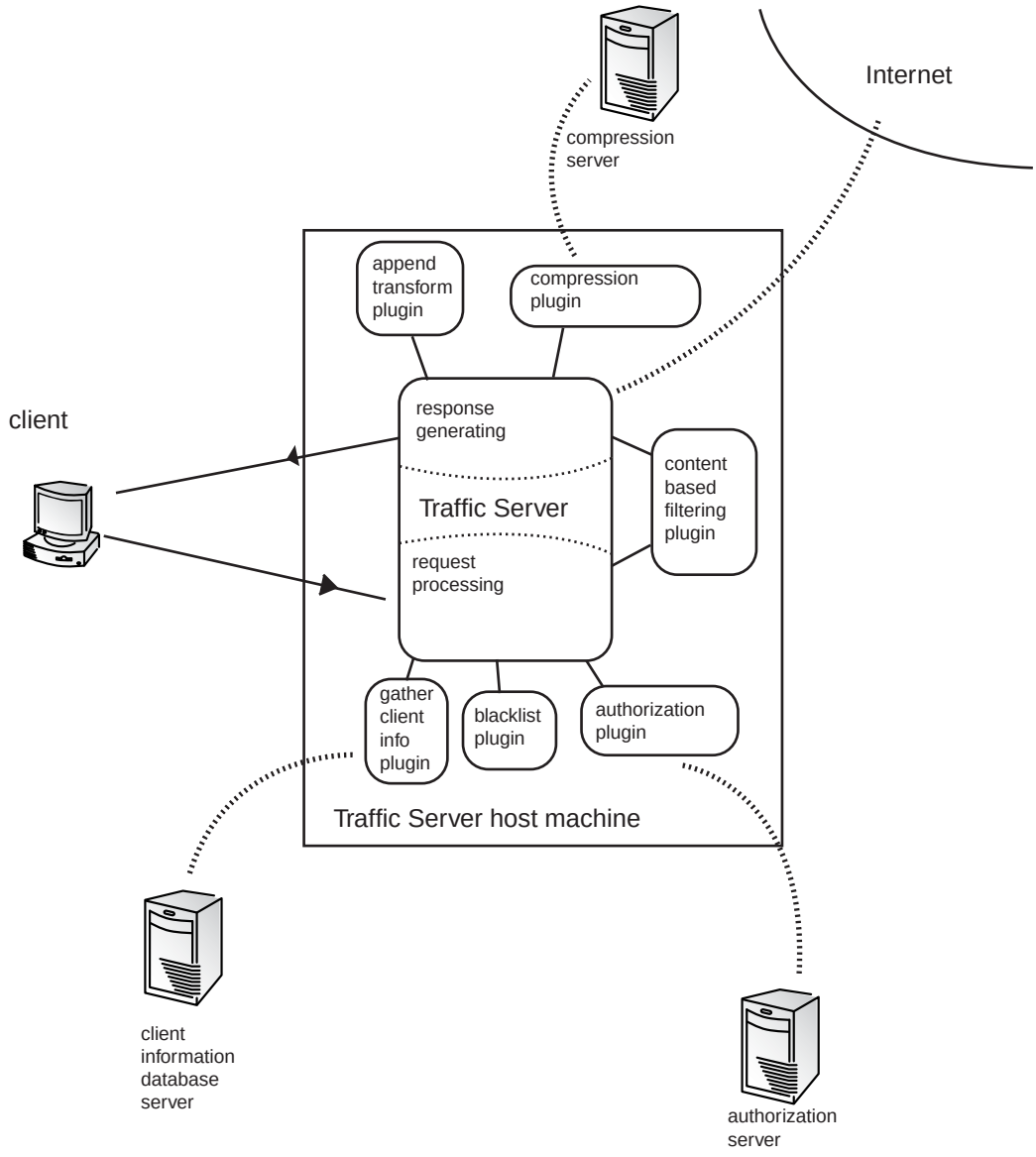


Figure 1 Possible Traffic Edge plugins

You can find basic examples of several of these plugins in the sample code provided with the SDK:

- `append-transform.c` adds text from a specified file to HTTP/text responses. This plugin is explained in detail in *"The append-transform plugin" on page 47*.

- The compression plugin in the figure communicates with a server that actually does the compression. The `server-transform.c` plugin shows how to open a connection to a transformation server, have the server do the transformation, and send transformed data back to the client. In `server-transform.c`, the transformation is null, but a compression or image translation plugin could be implemented in a similar way.
- `basic-auth.c` performs basic HTTP proxy authorization.
- `blacklist-1.c` reads blacklisted servers from a configuration file and denies client access to these servers. The plugin has a configuration interface where the Traffic Edge administrator can modify the list of blacklisted servers through the Traffic Manager GUI. This plugin is explained in detail in *"The Blacklist plugin" on page 31*.

Plugin loading

When Traffic Edge is first started, it consults the `plugin.config` file to determine the names of all the plugin shared libraries that need to be loaded. The `plugin.config` file also defines any arguments that are to be passed to each plugin's initialization function, *INKPluginInit*. The `records.config` file is used to define the path to each plugin shared library, described in *"Specifying the plugin's location" on page 19*.

Note The path for each of these files is `<root_dir>/config/`, where `<root_dir>` is the location where you installed Traffic Edge.

Plugin configuration

This sample `plugin.config` file contains a comment line, a blank line, and two plugin configurations:

```
# This is a comment line.

my-plugin.so www.junk.com www.trash.com www.garbage.com
some-plugin.so arg1 arg2 $proxy.config.http.cache.on
```

Each plugin configuration in the `plugin.config` file resembles a UNIX or DOS shell command.

limit on plugin.config entry lengths

Each line in `plugin.config` cannot exceed 1023 characters.

The first plugin configuration is for a plugin named `my-plugin.so` and contains three arguments that are to be passed to that plugin's initialization routine.

The second configuration is for a plugin named `some-plugin.so` and contains three arguments. The last argument, `$proxy.config.http.cache.on`, is actually a configuration variable. Traffic Edge will look up the specified configuration variable and substitute its value.

On the Windows NT version of Traffic Edge, the plugin shared library file is a `.dll` file. An example line in plugin.config would be the following:

```
nt_plugin.dll
```

multiple entries for the same plugin

Plugins with global variables should *not* appear more than once in `plugin.config`. For example, if you enter:

```
add-header.so header1
add-header.so header2
```

The second global variable, `header2`, would be used for both instances. A simple workaround is to give different instances of the same plugin different names, for example:

```
cp add-header.so add-header1.so
cp add-header.so add-header2.so
```

The following entries would have the desired result:

```
add-header1.so header1
add-header2.so header2
```

Configuration file rules

- Comment lines begin with a # and continue to the end of the line.
- Blank lines are ignored.
- Plugins are loaded and initialized by Traffic Edge in the order in which they appear in the `plugin.config` file.

Plugin initialization

Each plugin must define an initialization function named `INKPluginInit` that Traffic Edge invokes at the time the plugin is loaded. The `INKPluginInit` function is commonly used to read configuration information and register hooks for event notification.

The `INKPluginInit` function has two arguments:

- the `argc` argument represents the number of arguments defined in the `plugin.config` file for that particular plugin
- The `argv` argument is an array of pointers to the actual arguments defined in the `plugin.config` file for that plugin

See *"INKPluginInit" on page 142* for details about `INKPluginInit`.

A simple plugin

This section describes how you can write, compile, configure, and run a simple Traffic Edge plugin. Here are the steps you'll follow:

1. **Make sure that your plugin source code contains an `INKPluginInit` initialization function.**
1. Compile your plugin source code, creating a shared library.
2. Add an entry to the `plugin.config` file for your plugin.
3. Add the path to your plugin shared library to the `records.config` file.
4. Restart Traffic Edge.

hello world source

Shown below is the classic hello-world program implemented as a plugin using the Traffic Edge API.

```
#include <stdio.h>
#include "InkAPI.h"

void
INKPluginInit (int argc, const char *argv[])
{
    INKDebug ("debug-hello", "Hello World!\n");
}
```

In our simple hello-world example, *`INKPluginInit`* is the only function defined. This plugin does not use the `argc` or `argv` arguments. You can see more complex examples of `INKPluginInit` in the sample code provided with the SDK.

You need to make sure that the functions in your plugin are supported in your version of Traffic Edge. See *"Modified hello-world that checks Traffic Edge version" on page 20*.

Compiling your plugin

The process you use to compile a shared library will vary from platform to platform, so the Traffic Edge API includes makefile templates you can use to create shared libraries on all the supported Traffic Edge platforms.

Unix example

Assuming the sample program is stored in the file `hello-world.c`, you could use the following commands to building a shared library on Solaris using the GNU C compiler.

```
gcc -g -Wall -fPIC -o hello-world.o -c hello-world.c
gcc -g -Wall -shared -o hello-world.so hello-world.o
```

The first command compiles `hello-world.c` as Position Independent Code (PIC) and the second command links the single `hello-world.o` object file into the `hello-world.so` shared library.

Caution

Make sure that your plugin is **not** statically linked with system libraries.

HPUX example

Assuming the sample program is stored in the file `hello_world.c`, you could use the following commands to build a shared library on HPUX:

```
cc +z -o hello_world.o -c hello_world.c
ld -b -o hello_world.so hello_world.o
```

Compiling for Windows NT

Your PC must have the following software installed:

- Windows NT 4.0 SP4
- Microsoft Developer Studio 6.0

▼ **To compile a plugin for the Windows NT version of Traffic Edge**

1. Open `PlugIn.dsw` with Microsoft Visual C++ (MSVC++). The `dsw` file should be included in the SDK CD. Inside VC++, the sample plugins are listed as separate projects.
2. For each of the projects that need to be built, you need to tell VC++ where it can find the Traffic Edge library: `traffic_server.lib`. This library is in your NT Traffic Edge distribution.

   You might need to update the library lookup path. Use the following procedure:

▼ **To update the library lookup path**

1. Right-mouse-click on a project.
2. Select the **Settings...** option.
3. Click the **Link** tab on the dialog box.
4. Select **Input** in the combo-box.
5. Enter the library path in the **Additional library path:** text field

   Now you can build your plugin.

Updating the plugin.config file

Your next step is to tell Traffic Edge about the plugin by adding the following line to the `plugin.config` file. Since our simple plugin does not require any arguments, the following `plugin.config` will do nicely.

```
# a simple plugin.config for hello-world
hello-world.so
```

multiple plugins

Traffic Edge can accommodate multiple plugins. If several plugin functions are triggered by the same event, Traffic Edge will invoke each plugin's function in the order in which they were defined in the `plugin.config` file.

Specifying the plugin's location

All plugins must be located in the directory specified by the configuration variable `proxy.config.plugin.plugin_dir`, which is located in the `records.config` file. The directory can be specified as either an *absolute* or *relative* path.

If a *relative* path is used, the starting directory will be the Traffic Edge installation directory as specified in `/etc/traffic_server`. The default value is `config/plugins`, which tells Traffic Edge to use the directory `plugins` located in the same configuration directory as `records.config`. It is common to use the default directory.

Be sure to place your shared library `hello-world.so` inside the directory you have configured.

Restarting Traffic Edge

The last step is to start, or restart, Traffic Edge. Shown below is the output you would see after creating and loading your hello-world plugin.

```
# grep proxy.config.plugin.plugin_dir config/records.config
CONFIG proxy.config.plugin.plugin_dir STRING config/plugins
# ls config/plugins
hello-world.so*
# bin/traffic_server
[Mar 27 19:06:31.669] NOTE: updated diags config
[Mar 27 19:06:31.680] NOTE: loading plugin 'config/plugins/hello-world.so'
hello world
[Mar 27 19:06:32.046] NOTE: cache disabled (initializing)
[Mar 27 19:06:32.053] NOTE: cache enabled
[Mar 27 19:06:32.526] NOTE: Traffic Edge running
```

Note that in this example, the Traffic Edge notes are directed to the console by specifying `E` for `proxy.config.diags.output.note` in `records.config`. The second note shows the Traffic Edge attempting to load our hello-world plugin. The third line of Traffic Edge output is from your plugin.

Plugin Registration and Version Checking

You need to make sure that the functions in your plugin are supported in your version of Traffic Edge.

IMPORTANT Previous versions of Traffic Edge are named Traffic Server. Throughout this manual, Traffic Server, Traffic Server 3.0, Traffic Server 3.5, and Traffic Server 5.2 refer to previous versions of Traffic Edge. For version checking, Traffic Edge 1.5 is equivalent to Traffic Server 5.5.

Use the following interfaces:

- *INKPluginRegister, on page 142*
- *INKTrafficServerVersionGet, on page 143*

Modified hello-world that checks Traffic Edge version

The following version of hello-world registers the plugin and makes sure it is running with a compatible version of Traffic Edge.

```
#include <stdio.h>
#include "InkAPI.h"

int
check_ts_version() {

   const char* ts_version = INKTrafficServerVersionGet();
   int result = 0;

   if (ts_version) {
       int major_ts_version = 0;
       int minor_ts_version = 0;
       int patch_ts_version = 0;

       if (sscanf(ts_version, "%d.%d.%d", &major_ts_version,
                 &minor_ts_version, &patch_ts_version) != 3) {
```

```
            return 0;
        }

        /* Since this is an TS-SDK 2.0 plugin, we need at
           least Traffic Server 3.5.2 to run */
        if (major_ts_version > 3) {
            result = 1;
        } else if (major_ts_version == 3) {
            if (minor_ts_version > 5) {
                result = 1;
            } else if (minor_ts_version == 5) {
                if (patch_ts_version >= 2) {
                    result = 1;
                }
            }
        }
    }

    return result;
}

void
INKPluginInit (int argc, const char *argv[])
{
    INKPluginRegistrationInfo info;

    info.plugin_name = "hello-world";
    info.vendor_name = "MyCompany";
    info.support_email = "ts-api-support@MyCompany.com";

    if (!INKPluginRegister (INK_SDK_VERSION_2_0 , &info)) {
        INKError ("Plugin registration failed. \n");
    }

    if (!check_ts_version()) {
        INKError ("Plugin requires Traffic Server 3.5.2 or later\n");
        return;
    }

    INKDebug ("debug-hello", "Hello World!\n");
}
```

Naming conventions

The Traffic Edge API adheres to the following naming conventions:

- The `INK` prefix is used for all function and variable names defined in the Traffic Edge API. For example: `INK_EVENT_NONE`, `INKMutex` and `INKContCreate`.
- Enumerated values always appear in all uppercase letters. Examples: `INK_EVENT_NONE` and `INK_VC_CLOSE_ABORT`.
- Constant values are all upper case. Enumerated values can be seen as a subset of constants. Examples: `INK_URL_SCHEME_FILE` and `INK_MIME_FIELD_ACCEPT`.
- The names of defined types appear in mixed case. Examples: `INKHttpSsn` and `INKHttpTxn`.
- Function names are mixed case. Examples: `INKUrlCreate` and `INKContDestroy`.

- Function names use this subject-verb naming style: `INK-<subject>-<verb>`. The `<subject>` goes from the general to the specific. For example, the function to retrieve the password field (the specific subject) from a URL (the general subject) is `INKUrlPasswordGet`.This makes it easier to determine what a function does by reading its name.
- Common verbs like `Create`, `Destroy`, `Get`, `Set`, `Copy`, `Find`, `Retrieve`, `Insert`, `Remove` and `Delete` are used when appropriate.

CHAPTER 2

Creating Traffic Edge Plugins

This chapter provides a foundation for designing and writing plugins. Reading this chapter will help you understand:

- Inktomi's asynchronous event model, which is the design paradigm used throughout Traffic Edge. Plugins must also follow this design. It includes the callback mechanism for Traffic Edge to "wake up" your plugin and put it to work.
- Traffic Edge's HTTP processing—an overview of the HTTP state machine.
- How plugins can hook onto and modify or extend Traffic Edge's HTTP processing.
- A roadmap for writing plugins. An overview of the functionality provided by the Traffic Edge API.

The Asynchronous Event Model

Traffic Edge is a multi-threaded process. There are two main reasons why a server might use multiple threads:

- To take advantage of the concurrency available with multiple CPUs and multiple I/O devices.
- To manage concurrency from having many simultaneous client connections. For example a server could create one thread for each connection, allowing the operating system (OS) to control switching between threads.

Traffic Edge uses multiple threads for the first reason. But Traffic Edge does not use a separate OS thread per transaction because it would not be efficient when handling thousands of simultaneous connections.

Instead, Traffic Edge provides special event-driven mechanisms for efficiently scheduling work: the event system, and continuations. The event system is used to schedule work to be done on threads. A continuation is a passive, event-driven state machine that can do some work until it reaches a waiting point, and then sleep until it receives notification that conditions are right for doing more work. For instance, HTTP state machines (which handle HTTP transactions) are implemented as continuations.

Continuation objects are used throughout Traffic Edge. Some might live for the duration of the Traffic Edge process; others are created (perhaps by other continuations) for specific needs and then destroyed. Figure 2 shows how the major components of Traffic Edge interact. Traffic Edge has several processors, such as cache processor and net processor, which consolidate cache or network I/O tasks. Processors talk to the event system to schedule work on threads. An executing thread calls back a continuation by sending it an event. When a continuation receives an event, it wakes up, does some work, and either destroys itself or goes back to sleep waiting for the next event.

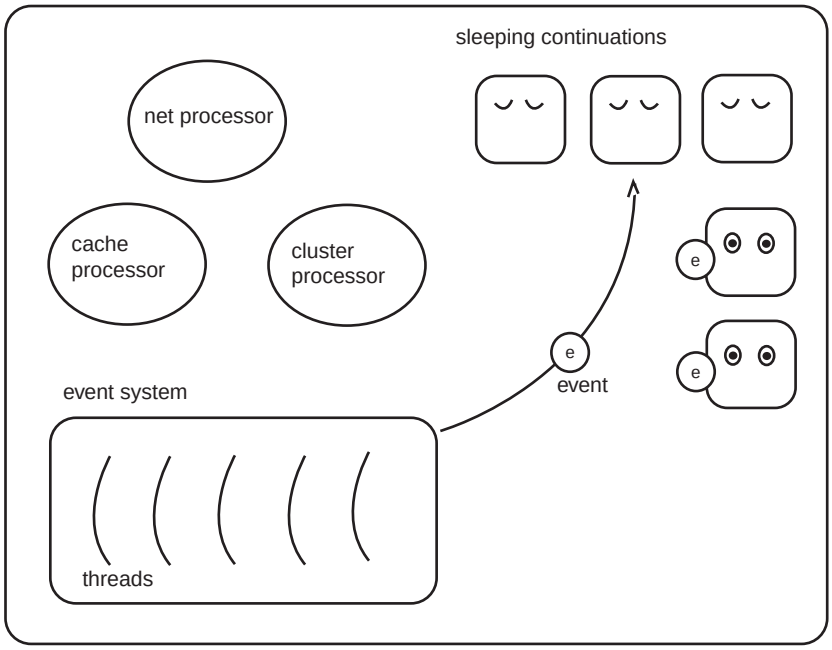


Figure 2 Traffic Edge internals

Plugins are typically implemented as continuations. All of the sample code plugins (except hello-world) are continuations that are created when Traffic Edge starts up; they wait for events that trigger them into activity.

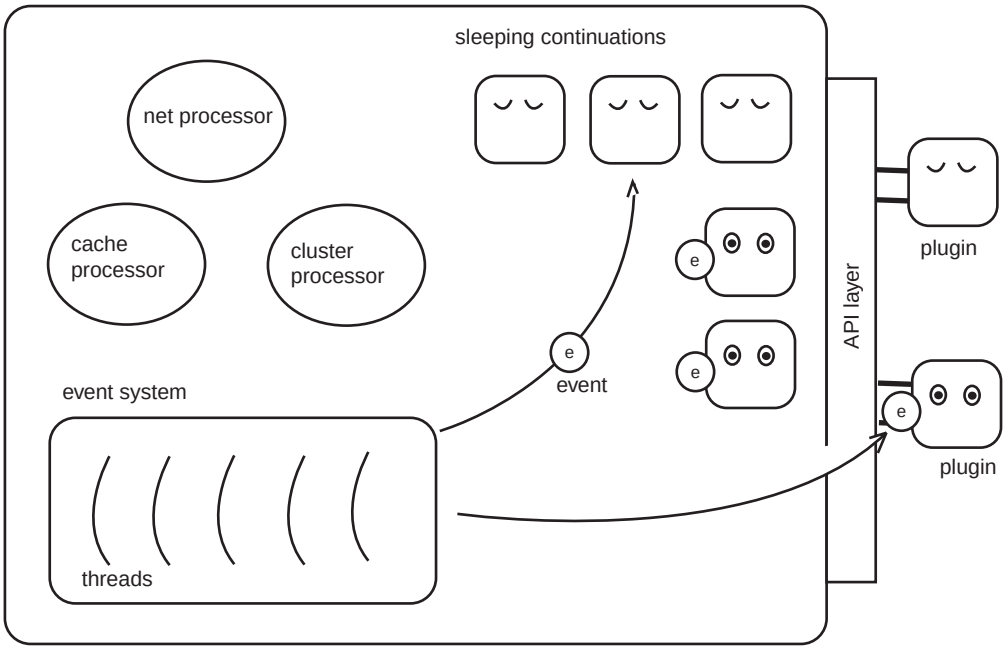


Figure 3 Traffic Edge with plugins

A plugin may consist of just one static continuation that is called whenever certain events happen. blacklist-1.c, basic-auth.c, and redirect-1.c are examples of such plugins. Or a plugin could dynamically create other continuations as needed. Transform plugins are built this way: a static parent continuation checks all transactions to see if any are transformable; when a transaction is transformable, the static continuation creates a type of continuation called a vconnection. The vconnection lives as long as it takes to complete the transform, and then destroys itself. You can see this design in all of the sample transform plugins. Plugins that support new protocols also have this architecture: a static continuation listens for incoming client connections, and creates transaction state machines to handle each protocol transaction.

When you write plugins, there are several ways to send events to continuations. For HTTP plugins, there is a "hook" mechanism that enables the Traffic Edge HTTP state machine to send your plugin wakeup calls when needed. Additionally, several Traffic Edge API functions trigger Traffic Edge sub-processes to send events to plugins: `INKContCall`, `INKVConnRead`, `INKCacheWrite`, and `INKMgmtUpdateRegister`, to name a few.

Traffic Edge HTTP State Machine

Traffic Edge does sophisticated HTTP caching and proxying. Its features include checking for alternates and document freshness, filtering, supporting cache hierarchies, and hosting. Traffic Edge handles thousands of client requests at a time, and each request is handled by an HTTP state machine. Traffic Edge's HTTP state machines follow a complex state diagram that includes all of the states required to support Traffic Edge's features. The Traffic Edge API provides hooks to a subset of these states, chosen for their relevance to plugins. You can view the API hooks and corresponding HTTP states in *"HTTP transaction state diagram" on page 66*.

This section goes through an example of how a plugin typically intervenes and extends Traffic Edge's processing of an HTTP transaction. Complete details about hooking on to Traffic Edge processes are provided in *"HTTP Hooks and Transactions" on page 65*.

HTTP transaction

An HTTP transaction consists of a client request for a web document and Traffic Edge's response. The response could be the requested web server content or it could be an error message. The content could come from the Traffic Edge cache or Traffic Edge might fetch it from the origin server. The following diagram shows some of the states of a typical transaction, highlighting the case where the content is served from the cache:

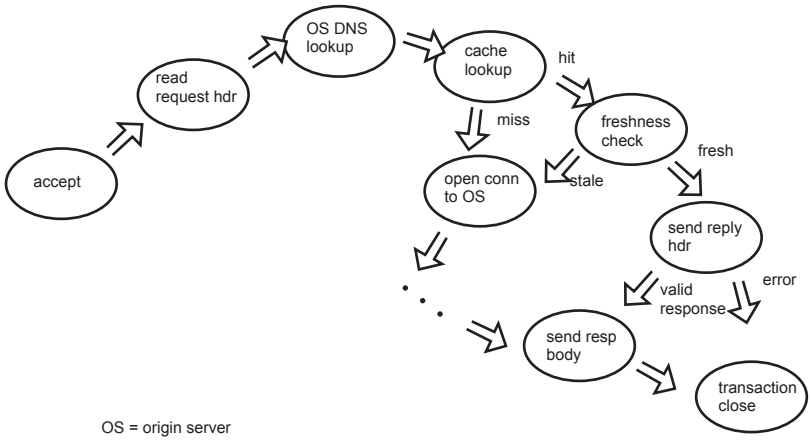


Figure 4 Simplified HTTP transaction

Traffic Edge accepts the client connection, reads the request headers, looks up the origin server's IP address, and looks for the requested content in the cache. If it's not in the cache, Traffic Edge opens a connection to the origin server and issues a request for the content. If the content is in the cache, Traffic Edge checks it for freshness. If it's fresh, Traffic Edge sends a reply header to the client. What Figure 4 does not show is that if there is an error at a any stage, the HTTP state machine jumps to the "send reply header" state and sends an error message. If the reply is an error, the transaction closes. If the reply is not an error, Traffic Edge sends the response content and then closes the transaction.

The Traffic Edge API supplies hooks that correspond to key stages in the HTTP state diagram. Figure 5 shows the API hooks that correspond to some of the states shown in Figure 4.

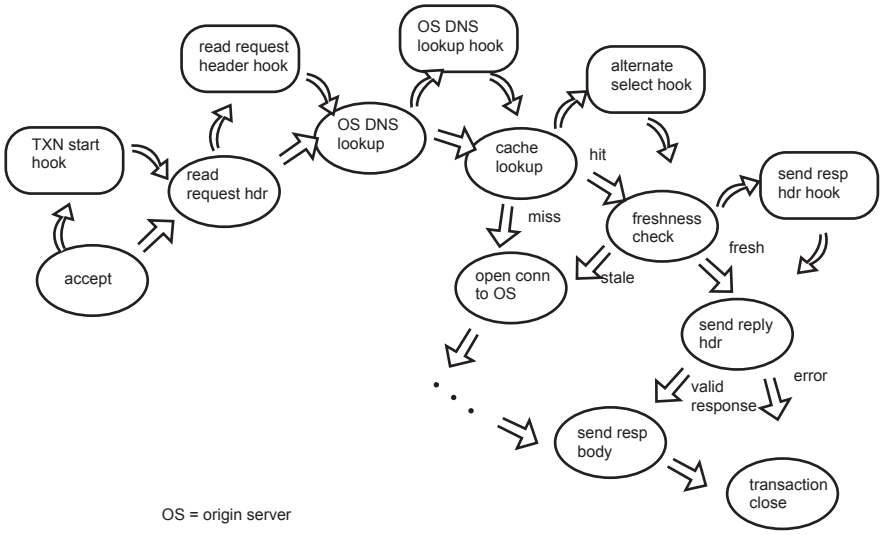


Figure 5 API hooks corresponding to states listed in Figure 4

You use hooks as triggers to start your plugin. The name of a hook reflects the Traffic Edge state that was just completed. So for example, the “OS DNS lookup” hook would wake up a plugin right after the origin server DNS lookup. For a plugin that requires the IP address of the requested origin server, this hook is the right one to use. The Blacklist plugin works this way, as shown in Figure 6.

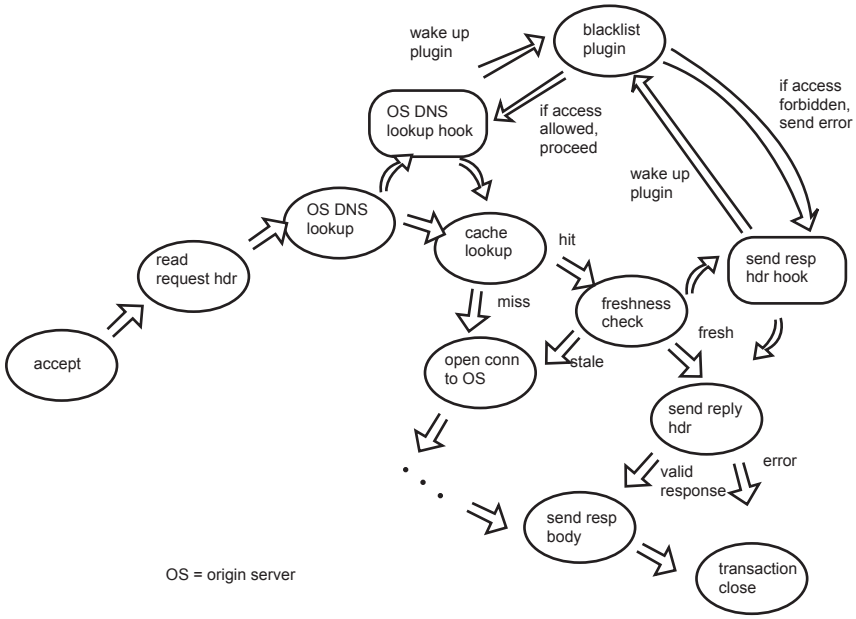


Figure 6 Blacklist plugin

Traffic Edge calls the Blacklist plugin right after the origin server DNS lookup. The plugin checks the requested host against a list of blacklisted servers, and if the request is allowed, the transaction proceeds. If the host is forbidden, the Blacklist plugin sends the transaction into an error state, and when the HTTP state machine gets to the “send reply header” state, it calls the Blacklist plugin to provide an error message to send to the client.

types of hooks

The Blacklist plugin’s hook to the “origin server DNS lookup” state is a *global hook*, meaning that the plugin is called for every HTTP transaction for which there is a DNS lookup event. The plugin’s hook to the “send reply header” state is a *transaction hook*, meaning that this hook is only invoked for specified transactions (in the Blacklist example, only for requests to blacklisted servers).

Several examples of setting up hooks are provided in the code example chapters, *“Header-Based Plugin Examples” on page 31*, and *“HTTP Transformation Plugins” on page 41*.

Header manipulation plugins, such as filtering, basic authorization, or redirects, usually have a global hook to the DNS lookup or the read request header states. Then if specific things need to be done to the transaction further on, the plugin adds itself to a transaction hook.

Transformation plugins require a global hook to check all transactions for transformability. Then they require a transform hook, which is a type of transaction hook specifically used for transforms.

Roadmap for creating plugins

So far this chapter has provided an overview of Traffic Edge's HTTP processing, API hooks, and the asynchronous event model. The next step is to understand the capabilities of the Traffic Edge API functions. These are very broad:

- HTTP header manipulation functions

  Obtain information about and manipulate HTTP headers, URLs, MIME headers.

- HTTP transaction functions

  Get information about and modify HTTP transactions (for example, get the client IP associated to the transaction; get the server IP; get parent proxy information)

- IO functions

  Manipulate vconnections (virtual connections, used for network and disk I/O).

- Network connection functions

  Open connections to remote servers.

- Statistics functions

  Define and compute statistics for your plugin's activity.

- Plugin management functions

  Create a web interface for your plugin (accessible through the Traffic Edge web interface). Control file installation. License your plugin.

- Traffic Edge management functions

  Obtain values of Traffic Edge configuration and statistics variables.

Here are some guidelines for creating a plugin:

1. Decide what you want your plugin to do, based on the capabilities of the API and Traffic Edge. The two main kinds of example plugins provided with SDK 5.2 are HTTP-based which include header-based plugins and response transform plugins,

and non-HTTP-based which includes a protocol plugin. These examples are discussed in the next three chapters.

2 Figure out where your plugin needs to hook on to Traffic Edge's HTTP processing. View the *"HTTP transaction state diagram" on page 66*.

3 Read *"Header-Based Plugin Examples" on page 31* to learn the basics of writing plugins: creating continuations, and setting up hooks. If you want to write a plugin that transforms data, read *"HTTP Transformation Plugins" on page 41*.

4 Figure out what parts of the Traffic Edge API you need to use, and read about the details of those APIs in the reference chapters in this manual.

5 Compile and load your plugin (see *"Getting Started" on page 13*).

6 Depending on your plugin's functionality, you might start testing it by issuing requests by hand, and checking for the desired behavior in Traffic Edge log files. See the *Traffic Edge Administrator's Guide* for information about Traffic Edge logs.

7 You can test the performance of Traffic Edge running with your plugin using SDKTest. You can also customize SDKTest to perform functional testing on your plugin. See the *Traffic Edge SDKTest User's Guide*.

CHAPTER 3 Header-Based Plugin Examples

Header-based plugins read or modify the headers of HTTP messages that Traffic Edge sends and receives. Reading this chapter will help you understand:

- Creating continuations for your plugins
- Adding global hooks
- Adding transaction hooks
- Working with HTTP header functions

The two sample plugins discussed in this chapter are `blacklist-1.c` and `basic-auth.c`.

Overview

Header-based plugins take actions based on the contents of HTTP request or response headers. Examples include filtering (on the basis of requested URL, or source IP address, or other request header), user authentication, or user redirection. These plugins have the following common elements:

- The plugin has a static parent continuation that scans all Traffic Edge headers (either request headers, response headers, or both).
- The plugin has a global hook. This allows the plugin to check all transactions to determine whether the plugin has to do something.
- Through the global hook, the plugin gets a handle to the transaction being processed.
- If the plugin needs to do something to transactions in specific cases, it sets up a transaction hook for a particular event.
- The plugin obtains client header information and does something based on it.

In the remainder of this chapter, you'll see how these components are implemented in SDK sample code.

The Blacklist plugin

The sample blacklisting plugin included in the Traffic Edge SDK is `blacklist_1.c`. This plugin checks every incoming HTTP client request against a list of blacklisted web sites. If the client requests a blacklisted site, the plugin returns an "access forbidden" message to the client. The flow of HTTP processing with the Blacklist plugin is illustrated in Figure 6, on page 27. This sample also contains a simple configuration management interface. It can read a list of blacklisted sites from a file, `blacklist.txt`, that can be updated by a Traffic Edge administrator. When the configuration file is updated, Traffic Edge sends an event to the plugin, waking it up to do some work.

Creating the parent continuation

You create the static parent continuation in the mandatory `INKPluginInit` function. This parent continuation effectively *is* the plugin: the plugin does work when this continuation receives an event from Traffic Edge. Traffic Edge passes the event as an argument to the continuation's handler function. When you create continuations, you must create and specify their handler functions.

You can specify an optional mutex lock when you create continuations. The mutex lock protects data shared by asynchronous processes. Traffic Edge has a multi-threaded design; if several threads try to access the same continuation's data, race conditions can occur.

Here is how the static parent continuation is created in `blacklist-1.c`:

```
void
INKPluginInit (int argc, const char *argv[])
{ ...
    INKCont contp;

    contp = INKContCreate (blacklist_plugin, NULL);
...
}
```

The handler function for the plugin is `blacklist_plugin`, and the mutex is null. The continuation handler function's job is to handle the events that are sent to it; accordingly, the `blacklist_plugin` routine consists of a switch statement that covers each of the events that might be sent to it:

```
static int
blacklist_plugin (INKCont contp, INKEvent event, void *edata)
{
    INKHttpTxn txnp = (INKHttpTxn) edata;

    switch (event) {
    case INK_EVENT_HTTP_OS_DNS:
        handle_dns (txnp, contp);
        return 0;
    case INK_EVENT_HTTP_SEND_RESPONSE_HDR:
        handle_response (txnp);
        return 0;
    case INK_EVENT_MGMT_UPDATE:
    read_blacklist ();
    return 0;
    default:
        break;
    }
    return 0;
}
```

When you write handler functions, you have to anticipate any events that might be sent to the handler by hooks or by other functions. In the Blacklist plugin, `INK_EVENT_OS_DNS` is

sent because of the global hook established in `INKPluginInit`; `INK_EVENT_HTTP_SEND_RESPONSE_HDR` is sent because the plugin contains a transaction hook (see *"Setting up a transaction hook" on page 34*), and `INK_EVENT_MGMT_UPDATE` is sent by Traffic Manager whenever there is a configuration change. See *"Setting Up UI Update Callbacks" on page 33*. It is good practice to have a default case in your switch statements.

Setting a Global Hook

Global hooks are always added in `INKPluginInit` using `INKHttpHookAdd`. The two arguments of `INKHttpHookAdd` are the hook ID and the continuation to call when processing the event corresponding to the hook. In blacklist-1.c, the global hook is added as follows:

```
INKHttpHookAdd (INK_HTTP_OS_DNS_HOOK, contp);
```

Where `INK_HTTP_OS_DNS_HOOK` is the ID for the origin server DNS lookup hook, and `contp` is the parent continuation created earlier.

This means that the Blacklist plugin is called at every origin server DNS lookup. When it is called, the handler function `blacklist_plugin` receives `INK_EVENT_HTTP_OS_DNS` and calls `handle_dns` to see if the request is forbidden.

Setting Up UI Update Callbacks

The Blacklist plugin must be called back whenever its configuration is changed by an administrator through the Traffic Manager UI. To get the interface working, you need an interface program (such as a CGI form) to display an interface and obtain configuration information, and a text file that the CGI program edits and the Blacklist plugin reads. The callback to the plugin is established in `INKPluginInit` by:

```
INKMgmtUpdateRegister (contp, "Inktomi Blacklist Plugin", "blacklist.cgi");
```

Where `contp` is the plugin's static parent continuation, `"Inktomi Blacklist Plugin"` is the name of the plugin as specified by the CGI form's `INK_PLUGIN_NAME` variable, and `"blacklist.cgi"` is the path to the plugin's interface program, relative to the Traffic Edge `plugins` directory. For more details see *"Setting up a plugin management interface" on page 131*.

Accessing the Transaction Being Processed

A continuation's handler function is of type `INKEventFunc`, and the prototype is as follows:

```
static int function_name (INKCont contp, INKEvent event, void *edata)
```

In general, the return value of the handler function is not used. The continuation argument is the continuation being called back, the event is the event being sent to the continuation, and the data pointed to by void *edata depends on the type of event. The data types for each event type are listed in *"Events and void * data" on page 111*.

The key here is that if the event is an HTTP transaction event, then the data passed to the continuation's handler is of type `INKHttpTxn` (a data type that represents HTTP transactions). Your plugin can then do things with the transaction. Here's how it looks in the Blacklist plugin's handler's code:

```
static int
blacklist_plugin (INKCont contp, INKEvent event, void *edata)
```

```
{
    INKHttpTxn txnp = (INKHttpTxn) edata;
    switch (event) {
        case INK_EVENT_HTTP_OS_DNS:
            handle_dns (txnp, contp);
            return 0;
        case INK_EVENT_HTTP_SEND_RESPONSE_HDR:
            handle_response (txnp);
            return 0;
        case INK_EVENT_MGMT_UPDATE:
            read_blacklist ();
            return 0;
        default:
            break;
    }
    return 0;

}
```

When, for example, the origin server DNS lookup event is sent, `blacklist_plugin` can call `handle_dns` and pass `txnp` as an argument.

Setting up a transaction hook

The Blacklist plugin sends "access forbidden" messages to clients if their requests are directed to blacklisted hosts. Therefore the plugin needs a transaction hook, so that it is called back when Traffic Edge's HTTP state machine reaches the "send response header" event. In the Blacklist plugin's `handle_dns` routine, the transaction hook is added as follows:

```
    INKMutexLock (sites_mutex);
    for (i = 0; i < nsites; i++) {
        if (strncmp (host, sites[i], host_length) == 0) {
            printf ("blacklisting site: %s\n", sites[i]);
            INKHttpTxnHookAdd (txnp,
                INK_HTTP_SEND_RESPONSE_HDR_HOOK,
                contp);
            INKHandleStringRelease (bufp, url_loc, host);
            INKHandleMLocRelease (bufp, hdr_loc, url_loc);
            INKHandleMLocRelease (bufp, INK_NULL_MLOC, hdr_loc);
            INKHttpTxnReenable (txnp, INK_EVENT_HTTP_ERROR);
        INKMutexUnlock (sites_mutex);
        return;
        }
    }

    INKMutexUnlock (sites_mutex);
```

```
    done:
        INKHttpTxnReenable (txnp, INK_EVENT_HTTP_CONTINUE);
}
```

This code fragment shows some interesting features. What's happening is that the plugin is comparing the requested site to the list of blacklisted sites. While the plugin is using the blacklist, it must acquire the mutex lock for the blacklist. This prevents configuration changes in the middle of a blacklisting operation. If the requested site is blacklisted, two things happen:

1. A transaction hook is added with `INKHttpTxnHookAdd`, so that the plugin is called back at the "send response header" event (the plugin sends an "access forbidden" message to the client). You can see that in order to add a transaction hook, you need a handle to the transaction being processed.
2. The transaction is reenabled using `INKHttpTxnReenable` with `INK_EVENT_HTTP_ERROR` as its event argument. Reenabling with an error event tells the HTTP state machine to stop the transaction and jump to the "send response header" state. Notice that if the requested site is not blacklisted, the transaction is reenabled with the `INK_EVENT_HTTP_CONTINUE` event.
3. The string and `INKMLoc` data stored in the marshal buffer bufp is released by `INKHandleStringRelease` and `INKHandleMLocRelease`. See *"Release marshal buffer handles" on page 88*. Release these handles before reenabling the transaction.

Reenable!

In general, whenever the plugin is doing something to a transaction, it must reenable the transaction when it is finished. Put another way, every time your handler function handles a transaction event, it must call `INKHttpTxnReenable` when it is finished.

Similarly, after your plugin handles session events (`INK_EVENT_HTTP_SSN_START` and `INK_EVENT_HTTP_SSN_CLOSE`) it must reenable the session with `INKHttpSsnReenable`.

but not twice!

Reenabling the transaction twice in the same plugin routine is a bad error.

Working with HTTP header functions

The Blacklist plugin examines the host header in every client transaction. This is done in the `handle_dns` routine, using *`INKHttpTxnClientIPGet`*, *`INKHttpHdrUrlGet`*, and *`INKUrlHostGet`*.

```
static void
handle_dns (INKHttpTxn txnp, INKCont contp)
{
    INKMBuffer bufp;
    INKMLoc hdr_loc;
    INKMLoc url_loc;
    const char *host;
    int i;

    if (!INKHttpTxnClientIPGet (txnp, &bufp, &hdr_loc)) {
        INKError ("couldn't retrieve client request header\n");
        goto done;
    }

    url_loc = INKHttpHdrUrlGet (bufp, hdr_loc);
```

```
    if (!url_loc) {
        INKError ("couldn't retrieve request url\n");
        INKHandleMLocRelease (bufp, INK_NULL_MLOC, hdr_loc);
        goto done;
    }

    host = INKUrlHostGet (bufp, url_loc, NULL);
    if (!host) {
        INKError ("couldn't retrieve request hostname\n");
        INKHandleMLocRelease (bufp, hdr_loc, url_loc);
        INKHandleMLocRelease (bufp, INK_NULL_MLOC, hdr_loc);
        goto done;
    }
```

To access the host header, the plugin first has to get the client request, then retrieve the URL portion, and then obtain the host header. See *"HTTP Headers" on page 83* for more information about these calls.

See *"Release marshal buffer handles" on page 88* for guidelines on using `INKHandleMLocRelease` and `INKHandleStringRelease`.

The Basic Authorization Plugin

The sample basic authorization plugin, `basic-auth.c`, checks for basic HTTP proxy authorization. In HTTP basic proxy authorization, client user names and passwords are contained in the `Proxy-Authorization` header. The password is encoded using base64 encoding. The plugin checks all incoming requests for the authorization header, user name and password. If the plugin does not find all of the these, it reenables with an error (effectively stopping the transaction) and adds a transaction hook to the send response header event.

Creating the plugin's parent continuation and global hook

The parent continuation and global hook are created as follows:

```
INKHttpHookAdd (INK_HTTP_OS_DNS_HOOK, INKContCreate (auth_plugin, NULL));
```

Implementing the handler and getting a handle to the transaction

The handler function for the plugin's parent continuation is implemented as follows:

```
static int
auth_plugin (INKCont contp, INKEvent event, void *edata)
{
    INKHttpTxn txnp = (INKHttpTxn) edata;
```

```
    switch (event) {
    case INK_EVENT_HTTP_OS_DNS:
        handle_dns (txnp, contp);
        return 0;
    case INK_EVENT_HTTP_SEND_RESPONSE_HDR:
        handle_response (txnp);
        return 0;
    default:
        break;
    }

    return 0;
}
```

Working with HTTP headers

The plugin checks all client request headers for the `Proxy-Authorization` MIME field, which should contain the user name and password.

The plugin's continuation handler, `auth-plugin`, calls `handle_dns` to check the `Proxy-Authorization` field.

The `handle_dns` routine uses `INKHttpTxnClientReqGet` and `INKMimeHdrFieldFind` to obtain the `Proxy-Authorization` field:

```
static void
handle_dns (INKHttpTxn txnp, INKCont contp)
{
    INKMBuffer bufp;
    INKMLoc hdr_loc;
    INKMLoc field_loc;
    const char *val;
    char *user, *password;

    if (!INKHttpTxnClientReqGet (txnp, &bufp, &hdr_loc)) {
        INKError ("couldn't retrieve client request header\n");
        goto done;
    }

    field_loc = INKMimeHdrFieldFind (bufp, hdr_loc,
                                     INK_MIME_FIELD_PROXY_AUTHORIZATION);
```

If the Proxy-Authorization field is present, the plugin checks that the authentication type is "Basic", and the user name and password are present and valid:

```
val = INKMimeHdrFieldValueStringGet (bufp, hdr_loc, field_loc, 0, &authval_length);
if (!val) {
        INKError ("no value in Proxy-Authorization field\n");
        INKHandleMLocRelease (bufp, hdr_loc, field_loc);
        INKHandleMLocRelease (bufp, INK_NULL_MLOC, hdr_loc);
        goto done;
```

```
}

if (strncmp (val, "Basic", 5) != 0) {
    INKError ("no Basic auth type in Proxy-Authorization\n");
    INKHandleStringRelease (bufp, field_loc, val);
    INKHandleMLocRelease (bufp, hdr_loc, field_loc);
    INKHandleMLocRelease (bufp, INK_NULL_MLOC, hdr_loc);
    goto done;
}

val += 5;
while ((*val == ' ') || (*val == '\t')) {
    val += 1;
}

user = base64_decode (val);
password = strchr (user, ':');
if (!password) {
    INKError ("no password in authorization information\n");
    INKfree (user);
    INKHandleStringRelease (bufp, field_loc, val);
    INKHandleMLocRelease (bufp, hdr_loc, field_loc);
    INKHandleMLocRelease (bufp, INK_NULL_MLOC, hdr_loc);
    goto done;
}
*password = '\0';
password += 1;

if (!authorized (user, password)) {
    INKError ("%s:%s not authorized\n", user, password);
    INKfree (user);
    INKHandleStringRelease (bufp, field_loc, val);
    INKHandleMLocRelease (bufp, hdr_loc, field_loc);
    INKHandleMLocRelease (bufp, INK_NULL_MLOC, hdr_loc);
    goto done;
}

INKfree (user);
INKHandleStringRelease (bufp, field_loc, val);
INKHandleMLocRelease (bufp, hdr_loc, field_loc);
INKHandleMLocRelease (bufp, INK_NULL_MLOC, hdr_loc);
INKHttpTxnReenable (txnp, INK_EVENT_HTTP_CONTINUE);
return;
```

Setting a transaction hook

If the request does not have the `Proxy-Authorization` field set to Basic authorization, or a valid user name and password, the plugin sends the `407 Proxy authorization required` status code back to the client.The client should then prompt the user for a user name and password, and resend the request.

In the handle_dns routine, the following lines handle the authorization error case:

```
done:
    INKHttpTxnHookAdd (txnp, INK_HTTP_SEND_RESPONSE_HDR_HOOK, contp);
    INKHttpTxnReenable (txnp, INK_EVENT_HTTP_ERROR);
```

If `handle_dns` does not find the `Proxy-Authorization` field set to Basic authorization, or a valid user name and password, it adds a `SEND_RESPONSE_HDR_HOOK` to the transaction being processed; this means that Traffic Edge will call the plugin back when sending the client response.

`handle_dns` reenables the transaction with `INK_EVENT_HTTP_ERROR`, which means that the plugin wants Traffic Edge to terminate the transaction.

When Traffic Edge terminates the transaction, it sends the client an error message. Because of the `SEND_RESPONSE_HDR_HOOK`, Traffic Edge calls the plugin back. The `auth-plugin` routine calls `handle_response` to send the client a 407 status code.

When the client resends the request with the `Proxy- Authorization` field, a new transaction begins.

`handle_dns` calls `base64_decode` to decode the user name and password.

`handle_dns` calls `authorized` to validate the user name and password. In this plugin, sample NT code is provided for password validation. Unix programmers can supply their own validation mechanism.

CHAPTER 4 HTTP Transformation Plugins

Transform plugins examine or transform HTTP message body content. For example, transform plugins can:

- Append text to HTML documents
- Compress images
- Do virus checking (on client POST data or server response data)
- Do content-based filtering (filter out HTML documents that contain certain terms or expressions)

In this chapter you can learn how to write transform plugins. The following examples are discussed in detail:

- *"The sample null transform plugin" on page 43*
- *"The append-transform plugin" on page 47*
- *"The sample buffered null transform plugin" on page 49*

Writing content transform plugins

Content transformation plugins transform HTTP response content (such as images or HTML documents), and HTTP request content such as client POST data. Because the data stream to be transformed is of variable length, these plugins must use a mechanism that passes data from buffer to buffer and checks to see if the end of the data stream is reached.

This mechanism is provided by virtual connections (*vconnections*) and virtual IO descriptors (VIOs).

A vconnection is an abstraction for a data pipe that allows its users to perform asynchronous reads and writes without knowing the underlying implementation. A *transformation* is a specific type of vconnection. A transformation connects an input data source and an output data sink; this feature enables it to view and modify all the data passing through it.

Transformations can be chained together, one after the other, so that multiple transformations can be performed on the same content. The vconnection type, `INKVConn`, is actually a subclass of `INKCont`, which means that vconnections (and transformations) are continuations. Vconnections and transformations can thus exchange events, informing one another (for example) that data is available for reading or writing, or that the end of a data stream is reached.

A VIO is a description of an in-progress IO operation. Every vconnection has an associated input VIO and an associated output VIO. When vconnections are transferring data to one another, one vconnection's input VIO is another vconnection's output VIO. A vconnection's input VIO is also called its write VIO because the input VIO refers to a write operation performed *on* the vconnection itself. Similarly, the outpt VIO is also called the

read VIO. For transformations, which are designed to pass data in one direction, you can picture the relationship between the transformation vconnection and its VIOs as follows:

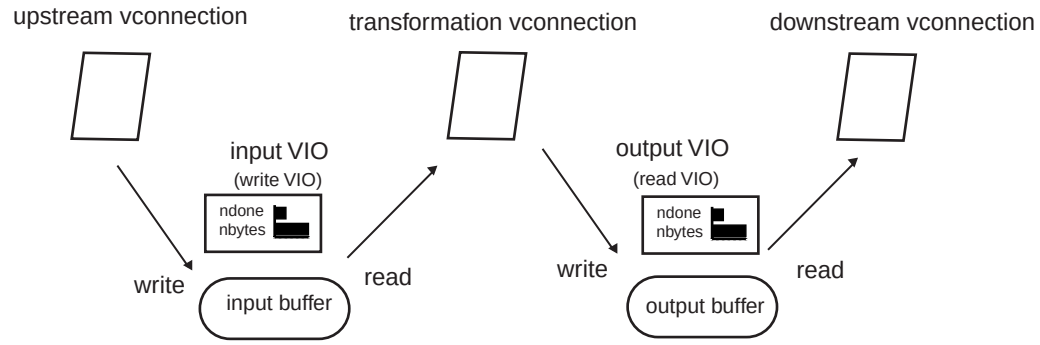


Figure 7 A transformation and its VIOs

Because the Traffic Edge API places transformations directly in the response or request data stream, the transformation vconnection is responsible only for reading the data from the input buffer, transforming it, and writing it to the output buffer. The upstream vconnection writes the incoming data to the transformation's input buffer. In Figure 7, the input VIO describes the progress of the upstream vconnection's write operation on the transformation, and the output VIO describes the progress of the transformation's write operation on the output (downstream) vconnection. The `nbytes` value in the VIO is the total number of bytes to be written. The ndone value is the current progress, the number of bytes written.

When writing a transformation plugin, you need to understand both implementing and using vconnections. The implementor's side refers to how to implement a vconnection that others can use. At minimum, a transform plugin creates a transformation that sits in the data stream and must be able to handle the events that the upstream and downstream vconnections send it. The user's side refers to how to use a vconnection to read or write data. Transformations output (write) data, at the very least.

Transformations

VIOs

A VIO or virtual IO is a description of an in progress IO operation. The VIO data structure is used by vconnection users to determine how much progress has been made on a particular IO operation and to re-enable an IO operation when it stalls due to buffer space. VIOs are used by vconnection implementors to determine the buffer for an IO operation, to determine how much work to do on the IO operation and to determine which continuation to call back when progress on the IO operation is made.

The `INKVIO` data structure itself is opaque, but it might have been defined as follows:

```
typedef struct {
    INKCont continuation;
    INKVConn vconnection;
    INKIOBufferReader reader;
    INKMutex mutex;
```

```
    int nbytes;
    int ndone;
} *INKVIO;
```

IO buffers

The IO buffer data structure is the building block of the *vconnection* abstraction. An IO buffer is composed of a list of buffer blocks which in turn point to buffer data. Both the buffer block (`INKIOBufferBlock`) and buffer data (`INKIOBufferData`) data structures are reference counted so that they can reside in multiple buffers at the same time. This makes it extremely efficient to copy data from one IO buffer to another using `INKIOBufferCopy` since Traffic Edge only needs to copy pointers and adjust reference counts appropriately and not actually copy any data.

The IO buffer abstraction provides for a single writer and multiple readers. In order for the readers to have no knowledge of each other, they manipulate IO buffers through the `INKIOBufferReader` data structure. Since only a single writer is allowed, there is no corresponding `INKIOBufferWriter` data structure. The writer simply modifies the IO buffer directly.

The sample null transform plugin

This section provides a step-by-step description of what the null transform plugin does, along with sections of the code that apply. For context, you can find each code snippet in the complete source code. Some of the error checking details are left out; to give the description a step-by-step flow, only the highlights of the transform are included.

Here is an overview of the null transform plugin:

1 Gets a handle to HTTP transactions.

```
void
INKPluginInit (int argc, const char *argv[]) {
    INKHttpHookAdd (INK_HTTP_READ_RESPONSE_HDR_HOOK,
              INKContCreate (transform_plugin, NULL)); }
```

With this `INKPluginInit` routine, the plugin is called back every time Traffic Edge reads a response header.

2 Checks to see if the transaction response is transformable.

```
static int transform_plugin (INKCont contp, INKEvent event, void *edata) {
    INKHttpTxn txnp = (INKHttpTxn) edata;
    switch (event) {
        case INK_EVENT_HTTP_READ_RESPONSE_HDR:
        if (transformable (txnp)) {
            transform_add (txnp);}
```

The default behavior for transformations is to cache the transformed content. (You can tell Traffic Edge to cache untransformed content, if you want). Therefore, only responses received directly from an origin server need be transformed. Objects served from the cache are already transformed. To determine whether the response is from the origin server, the routine `transformable` checks the response header for the "`200 OK`" server response.

```
static int transformable (INKHttpTxn txnp)
{
   INKMBuffer bufp;
   INKMLoc hdr_loc;
   INKHttpStatus resp_status;

   INKHttpTxnServerRespGet (txnp, &bufp, &hdr_loc);

   if (INK_HTTP_STATUS_OK == (resp_status =
      INKHttpHdrStatusGet (bufp, hdr_loc)) ) {
      return 1;
   } else {
      return 0;
   }
}
```

3 If the response is transformable, the plugin creates a transformation vconnection that gets called back when the response data is ready to be transformed (as it is streaming from the origin server).

```
static void transform_add (INKHttpTxn txnp)
{
   INKVConn connp;
   connp = INKTransformCreate (null_transform, txnp);
   INKHttpTxnHookAdd (txnp, INK_HTTP_RESPONSE_TRANSFORM_HOOK, connp);
}
```

The previous code fragment shows that the handler function for the transformation vconnection is `null_transform`.

4 Get a handle to the output vconnection (that receives data from the tranformation).

```
output_conn = INKTransformOutputVConnGet (contp);
```

5 Get a handle to the input VIO. (See the `handle_transform` function.)

```
input_vio = INKVConnWriteVIOGet (contp);
```

This is so that the transformation can get information about the upstream vconnection's write operation to the input buffer.

6 Initiate a write to the output vconnection of the specified number of bytes. When the write is initiated, the transformation expects to receive `WRITE_READY`, `WRITE_COMPLETE`, or `ERROR` events from the output vconnection.

See the `handle_transform` function for the following code fragment:

```
data->output_vio = INKVConnWrite (output_conn, contp,
          data->output_reader, INKVIONBytesGet (input_vio));
```

7 Copy data from the input buffer to the output buffer. See the `handle_transform` function for the following code fragment:

```
INKIOBufferCopy (INKVIOBufferGet (data->output_vio),
          INKVIOReaderGet (input_vio), towrite, 0);
```

8 Tell the input buffer that the transformation has read the data. See the `handle_transform` function for the following code fragment:

```
INKIOBufferReaderConsume (INKVIOReaderGet (input_vio), towrite);
```

9 Modify the input VIO to tell it how much data has been read (increase the value of `ndone`). See the `handle_transform` function for the following code fragment:

```
INKVIONDoneSet (input_vio, INKVIONDoneGet (input_vio) + towrite);
```

10 If there is more data left to read (if `ndone` < `nbytes`), the handle_transform function wakes up the downstream vconnection with a reenable and wakes up the upstream vconnection by sending it `WRITE_READY`:

```
if (INKVIONTodoGet (input_vio) > 0) {
   if (towrite > 0) {
      INKVIOReenable (data->output_vio);

      INKContCall (INKVIOContGet (input_vio),
            INK_EVENT_VCONN_WRITE_READY, input_vio);
   }
} else {
```

The process of passing data through the transformation is illustrated in the following diagram. The downstream vconnections send `WRITE_READY` events when they need more data, and when data is available the upstream vconnections reenable the downstream vconnections. The `INKVIOReenable` function, in this instance, sends `INK_EVENT_IMMEDIATE`.

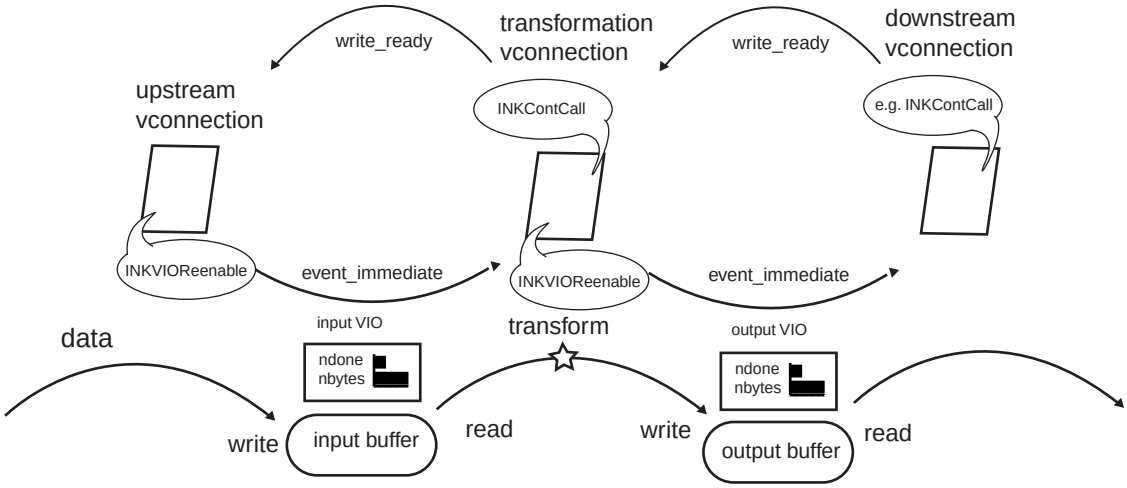


Figure 8 Passing data through a transformation

11 If the `handle_transform` function finds that there is no more data to read, it sets `nbytes` to `ndone` on the output (downstream) VIO, and wakes up the output vconnection with a reenable. It then triggers the end of the write operation from the upstream vconnection by sending the upstream vconnection a `WRITE_COMPLETE` event.

```
} else {

INKVIONBytesSet (data->output_vio, INKVIONDoneGet (input_vio));
INKVIOReenable (data->output_vio);
INKContCall (INKVIOContGet (input_vio),
    INK_EVENT_VCONN_WRITE_COMPLETE, input_vio);
}
```

When the upstream vconnection receives the `WRITE_COMPLETE` event, it will probably shut down the write operation.

12 Similarly, when the downstream vconnection has consumed all of the data, it sends the transformation a `WRITE_COMPLETE` event. The transformation handles this event with a shut down (the transformation shuts down the write operation to the downstream vconnection). See the `null_plugin` function for the following code fragment:

```
case INK_EVENT_VCONN_WRITE_COMPLETE:
```

```
INKVConnShutdown (INKTransformOutputVConnGet (contp), 0, 1);
break;
```

The following diagram illustrates the flow of events:

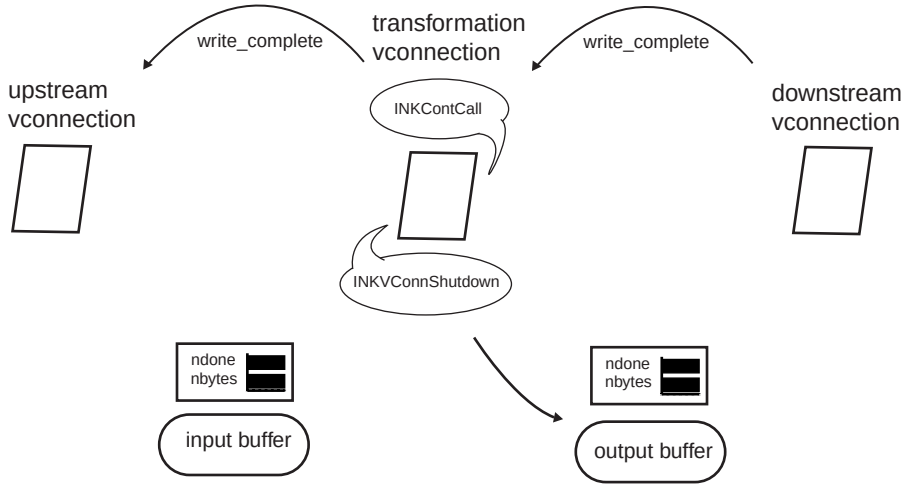


Figure 9 Ending the transformation

The append-transform plugin

The append-transform plugin appends text to the body of an HTTP response. It obtains this text from a file. The name of the file containing the append text is a parameter you specify in `plugin.config`, as follows:

`append-transform.so` *`path/to/file`*

The append-transform plugin is based on `null-transform.c`. The only difference is that after the plugin feeds the document through the transformation, it adds text to the response.

Here is a list of the functions in `append-transform.c`, in the order they appear in the source code, with a description of what the function does:

- `my_data_alloc`

  Allocates and initializes a `MyData` structure. The plugin defines a struct, `MyData`, as follows:

```
typedef struct {
INKVIO output_vio;
INKIOBuffer output_buffer;
INKIOBufferReader output_reader;
int append_needed;
```

```
} MyData;
```

The `MyData` structure is used to represent data that the transformation (vconnection) needs. The transformation's data pointer is set to a `MyData` pointer using `INKContDataSet` in the `handle_transform` routine.

- `my_data_destroy`

  Destroys objects of type `MyData`. The `append_transform` routine (see below) calls `my_data_destroy` when th e transformation is complete, to deallocate the transformation's data.

- `handle_transform`

  This function does the actual data transformation. The transformation is created in `transform_add` (see below). handle_transform is called by `append_transform`.

- `append_transform`

  This is the handler function for the transformation vconnection created in `transform_add`. It is the implementation of the vconnection.

  - If the transformation vconnection has been closed, `append_transform` calls `my_data_destroy` to destroy the vonnection
  - If `append_transform` receives an error event, it calls back the continuation to let it know it has completed the write operation
  - If it receives a `WRITE_COMPLETE` event, it shuts down the write portion of its vconnection
  - If it receives a `WRITE_READY` or any other event (such as `INK_HTTP_RESPONSE_TRANSFORM_HOOK`), it calls `handle_transform` to attempt to transform more data

- `transformable`

  The plugin transforms only documents that have a content type of `text/html`. This function examines the `Content-Type` MIME header field in the response header; if the value of the MIME field is `text/html`, the function returns 1. Otherwise, it returns zero.

- `transform_add`

  Creates the transformation for the current transaction, and sets up a transformation hook. The handler function for the transformation is `append_transform`.

- `transform_plugin`

  This is the handler function for the main continuation for the plugin. Traffic Edge calls this function whenever it reads an HTTP response header. `transform_plugin` does the following:

  - Gets a handle to the HTTP transaction being processed
  - Calls `transformable` to determine whether the response document content is of type `text/html`
  - If the content is transformable, calls `transform_add` to create the transformation
  - Calls `INKHttpTxnReenable` to continue the transaction

- `load`

  Opens the file containing the text to be appended, and loads the contents of the file into an `INKIOBuffer` called `append_buffer`.

- `INKPluginInit`

  Does the following:

  - Checks to make sure that the required configuration information (the append text filename) is entered in `plugin.config` correctly.
  - If there is a filename, `INKPluginInit` calls `load` to load the text.
  - Creates a continuation for the plugin. The handler for this continuation is `transform_plugin`.
  - Adds the plugin's continuation to `INK_HTTP_READ_RESPONSE_HDR_HOOK`. In other words, sets up a callback of the plugin's continuation when Traffic Edge reads HTTP response headers.

The sample buffered null transform plugin

The buffered null transform, `bnull-transform.c`, reads the response content into a buffer and then writes the full buffer out to the client. Many examples of transformations, such as compression, require you to gather the full response content in order to perform the transformation.

The buffered null transform uses a state variable to keep track of when it is (a) reading data into the buffer and (b) writing the data from the buffer to the downstream vconnection.

The following is a step-by-step walk through the buffered null transform:

1 Gets a handle to HTTP transactions.

```
void
INKPluginInit (int argc, const char *argv[]) {
   INKHttpHookAdd (INK_HTTP_READ_RESPONSE_HDR_HOOK,
             INKContCreate (transform_plugin, NULL)); }
```

With this `INKPluginInit` routine, the plugin is called back every time Traffic Edge reads a response header.

2 Checks to see if the transaction response is transformable.

```
static int transform_plugin (INKCont contp, INKEvent event, void *edata) {
   INKHttpTxn txnp = (INKHttpTxn) edata;
   switch (event) {
      case INK_EVENT_HTTP_READ_RESPONSE_HDR:
      if (transformable (txnp)) {
```

```
        transform_add (txnp);}
```

The default behavior for transformations is to cache the transformed content. (You can tell Traffic Edge to cache untransformed content, if you want). Therefore, only responses received directly from an origin server need be transformed. Objects served from the cache are already transformed. To determine whether the response is from the origin server, the routine `transformable` checks the response header for the "`200 OK`" server response.

```
static int transformable (INKHttpTxn txnp)
{
    INKMBuffer bufp;
    INKMLoc hdr_loc;
    INKHttpStatus resp_status;

    INKHttpTxnServerRespGet (txnp, &bufp, &hdr_loc);

    if(INK_HTTP_STATUS_OK==
       (resp_status=INKHttpHdrStatusGet(bufp,hdr_loc)))
       {
          return 1;
       }
       else {
          return 0;
       }
}
```

3 If the response is transformable, the plugin creates a transformation vconnection that gets called back when the response data is ready to be transformed (as it is streaming from the origin server).

```
static void transform_add (INKHttpTxn txnp)
{
    INKVConn connp;
    connp = INKTransformCreate (bnull_transform, txnp);
    INKHttpTxnHookAdd (txnp, INK_HTTP_RESPONSE_TRANSFORM_HOOK, connp);
}
```

The previous code fragment shows that the handler function for the transformation vconnection is `bnull_transform`.

4 The `bnull_transform` function has to handle `ERROR`, `WRITE_COMPLETE`, `WRITE_READY`, and `IMMEDIATE` events. If the transform is just beginning, the event received is probably `IMMEDIATE`. The `bnull_transform` function calls `handle_transform` to handle `WRITE_READY` and `IMMEDIATE`.

5 The `handle_transform` function examines the data parameter for the continuation passed to it (the continuation passed to handle_transform is the transformation vconnection). The data structure keeps track of two states: copying the data into the

buffer (STATE_BUFFER_DATA) and writing the contents of the buffer to the output vconnection (STATE_OUTPUT_DATA).

If the state is STATE_BUFFER_DATA, handle_transform calls handle_buffering to copy data into the buffer.

6 Get a handle to the input VIO. (See the handle_buffering function.)

```
input_vio = INKVConnWriteVIOGet (contp);
```

This is so that the transformation can get information about the upstream vconnection's write operation to the input buffer.

7 Copy data from the input buffer to the output buffer. See the handle_buffering function for the following code fragment:

```
INKIOBufferCopy (data->output_buffer,
       INKVIOReaderGet (write_vio), towrite, 0);
```

8 Tell the input buffer that the transformation has read the data. See the handle_buffering function for the following code fragment:

```
INKIOBufferReaderConsume (INKVIOReaderGet (write_vio), towrite);
```

9 Modify the input VIO to tell it how much data has been read (increase the value of ndone). See the handle_buffering function for the following code fragment:

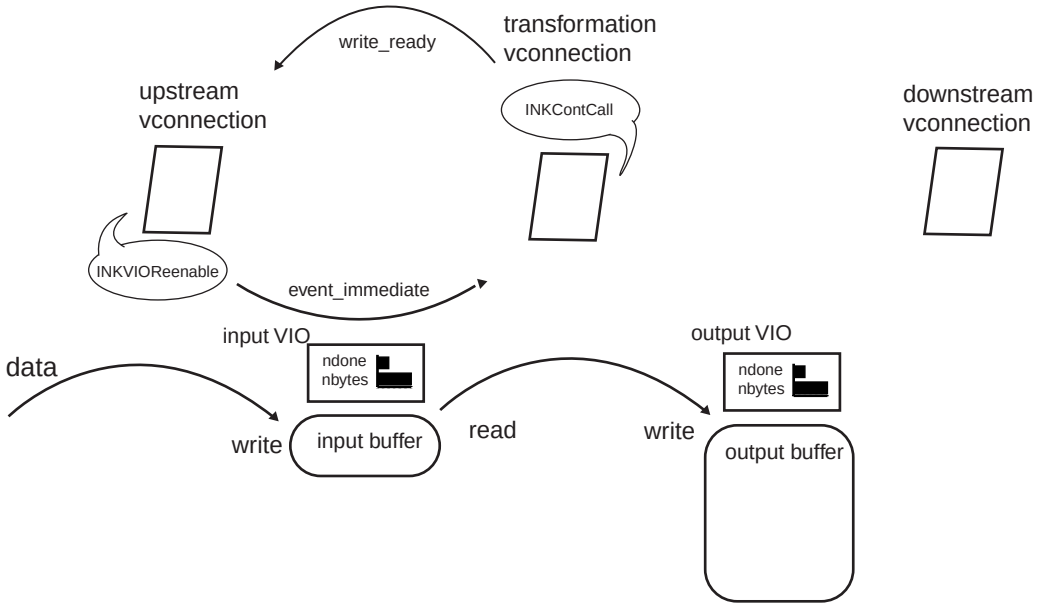
```
INKVIONDoneSet (write_vio, INKVIONDoneGet (write_vio) + towrite); }
```

10 If there is more data left to read (if ndone < nbytes), the handle_buffering function wakes up the upstream vconnection by sending it WRITE_READY:

```
if (INKVIONTodoGet (write_vio) > 0) {
   if (towrite > 0) {
      INKContCall (INKVIOContGet (write_vio),
                INK_EVENT_VCONN_WRITE_READY, write_vio);
   }
} else {
```

The process of passing data through the transformation is illustrated in the following diagram. The transformation sends WRITE_READY events when it needs more data, and when data is available the upstream vconnection reenables the transformation with an IMMEDIATE event.

Figure 10 Reading data into the buffer (the STATE_BUFFER_DATA state)



11 When the data is read into the output buffer, the handle_buffering function sets the state of the transformation's data structure to `STATE_OUTPUT_DATA`. and calls the upstream vconnection back with the `WRITE_COMPLETE` event.

```
data->state = STATE_OUTPUT_DATA;
INKContCall (INKVIOContGet (write_vio),
    INK_EVENT_VCONN_WRITE_COMPLETE, write_vio);
```

12 The upstream vconnection will probably shut down the write operation when it receives the `WRITE_COMPLETE` event. The handler function of the transformation, `bnull_transform`, will receive an `IMMEDIATE` event, and call the `handle_transform` function. This time, the state is `STATE_OUTPUT_DATA`, so `handle_transform` calls `handle_output`.

13 The `handle_output` function gets a handle to the output vconnection:

```
output_conn = INKTransformOutputVConnGet (contp);
```

14 The `handle_output` function writes the buffer to the output vconnection:

```
data->output_vio =
INKVConnWrite (output_conn, contp, data->output_reader,
INKIOBufferReaderAvail (data->output_reader) );
```

The following diagram illustrates the write to the output vconnection:

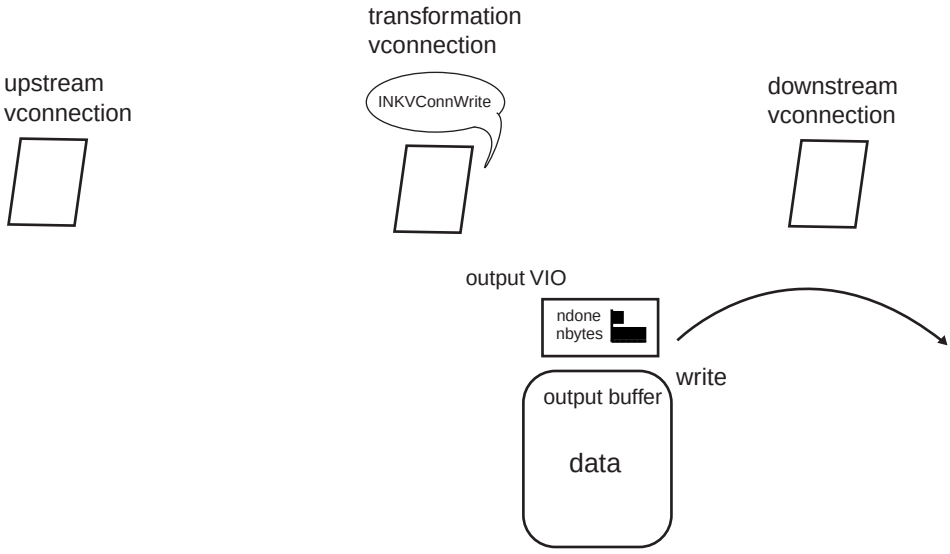


Figure 11 Writing the buffered data to the output vconnection

CHAPTER 5 New Protocol Plugins

The new protocol APIs allow you to extend Traffic Edge to be a web proxy for any protocol. This chapter describes the new protocol APIs and plugins that support new protocols. It goes through sample Protocol plugin code in detail. The sample Protocol plugin supports a very simple artificial HTTP-like protocol.

This chapter contains the following sections:

- *"About the sample protocol" on page 55*

  Gives the state diagram and header structure of the artificial protocol. Describes what the supporting plugin has to do.

- *"Protocol plugin structure" on page 58*

  In depth explanation of the Protocol plugin. Starts with overall architecture, and describes how to write continuations as state machines. Ends with a walk-through of the Protocol plugin code as it processes a transaction.

About the sample protocol

The sample protocol allows a client to ask a server for a file. Clients send requests to a specific Traffic Edge port (specified in `plugin.config`). The requests look like the following:

`server_name file_name``\n\n`

With the Protocol plugin, Traffic Edge can accept these requests, parse them, and act as a proxy cache (requesting the file from the origin server on the client's behalf, and storing copies of the response messages in the cache).

The Protocol plugin is a state machine that flows through the states illustrated in Figure 12. The figure shows the steps that Traffic Edge and the Protocol plugin go through to support the sample protocol. In words, Traffic Edge and the Protocol plugin must:

- listen for and accept client connections (on the accept port specified in `plugin.config`)
- read incoming client requests
- look up the requested content in the Traffic Edge cache
- if the request is a cache hit, serve the content from the cache (this simple example does not do freshness checking)
- if the request is a cache miss, open a connection to the origin server (on the server port specified in `plugin.config`)
- forward the request to the origin server
- receive the origin server response

- cache the response and send it on to the client

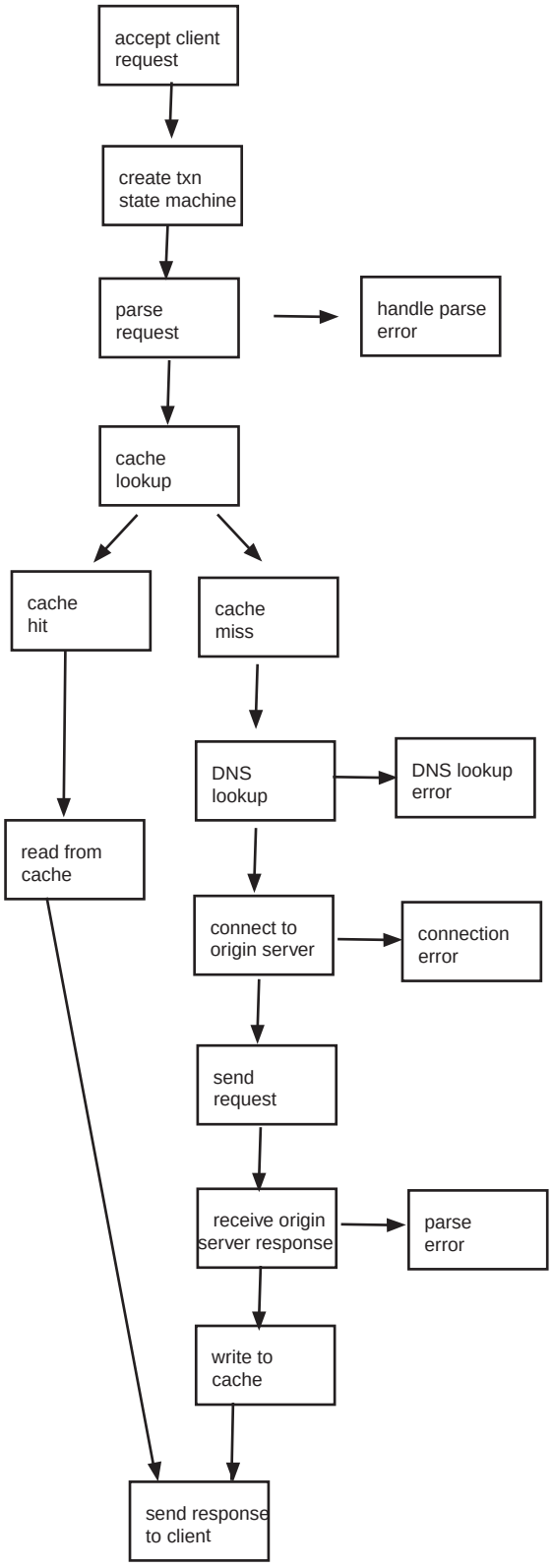


Figure 12 Sample protocol state diagram

Protocol plugin structure

To see how the Protocol plugin works, you need to understand a couple of big pictures.

This section assumes you are familiar with the concepts of continuation, Traffic Edge's asynchronous event model, and basic Traffic Edge plugin structure. If not, see *"Getting Started" on page 13* and *"Creating Traffic Edge Plugins" on page 23*.

Continuations in the Protocol plugin

The Protocol plugin creates a static continuation that is an "accept" state machine, a state machine whose job is to accept client connections on the appropriate port. When Traffic Edge accepts a net connection from a client on that port, the accept state machine is activated and it creates a new continuation, a transaction state machine. The accept state machine creates one transaction state machine for each transaction (a transaction consists of a client request and Traffic Edge's response). Each transaction state machine lives until the transaction completes, and then it is destroyed. If the client's request for content is a cache miss, a transaction state machine might have to open a connection to the origin server. This is illustrated in Figure 13.

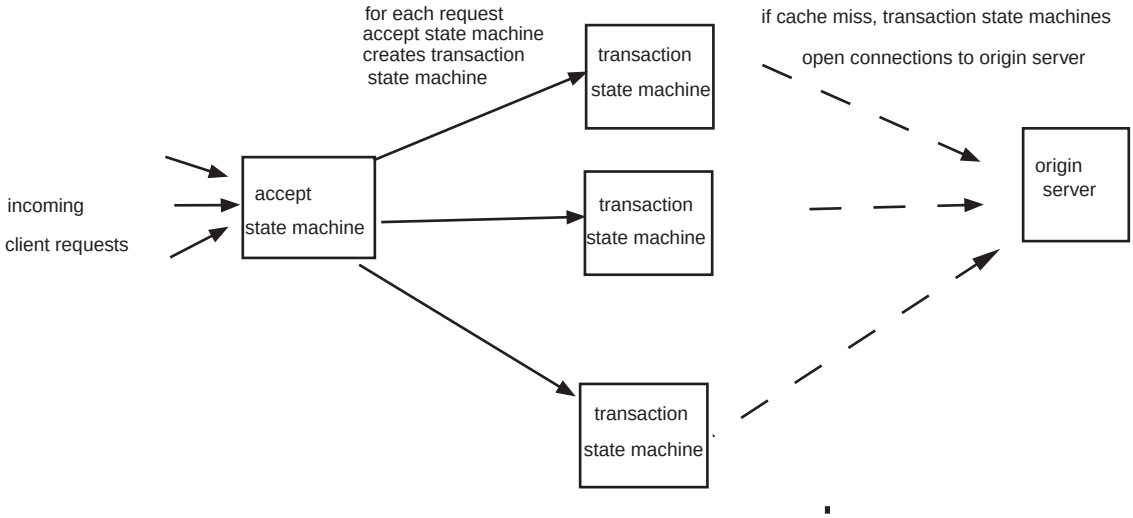


Figure 13 Protocol plugin overview

Now you can see the first steps in writing this Protocol plugin: in `INKPluginInit`, you must create a continuation that listens for net connections on the client port specified in `plugin.config` (this continuation is the accept state machine).

Here is a summary of the continuations implemented for the Protocol plugin:

- An accept state machine that listens for client connections, and creates transaction state machines whenever Traffic Edge accepts a new client connection. The accept state machine lives as long as Traffic Edge is running.
- Transaction state machines that read client requests, process them, and are destroyed when the transaction is done.

Event flow

To understand how to implement the rest of the Protocol plugin you need to understand the flow of events that takes place in the course of a transaction. Unlike HTTP transaction plugins, this plugin must read data from network connections and read and write data to the Traffic Edge cache. This means that its continuations do not receive HTTP state machine events; they receive events from Traffic Edge's processor subsystems.

For example, the accept state machine is activated by an `INK_EVENT_NET_ACCEPT` event from Traffic Edge's Net Processor. The handler function for the accept state machine must be able to handle that event.

The transaction state machines are activated when the client connection receives incoming request data. The Net Processor notifies the transaction state machine of incoming data. The transaction state machine reads the data, and then when it is done, initiates a cache lookup of the requested file. When the cache lookup completes, the transaction state machine is activated by the Traffic Edge Cache Processor.

If the transaction state machine has to open a connection to the origin server to fetch content (in the case of a cache miss), the transaction state machine initiates a DNS lookup of the server name. The transaction state machine is activated by a DNS lookup event from the Traffic Edge Host Database Processor.

If the transaction has to connect to the origin server, the transaction state machine initiates a net connection and waits for an event from Net Processor.

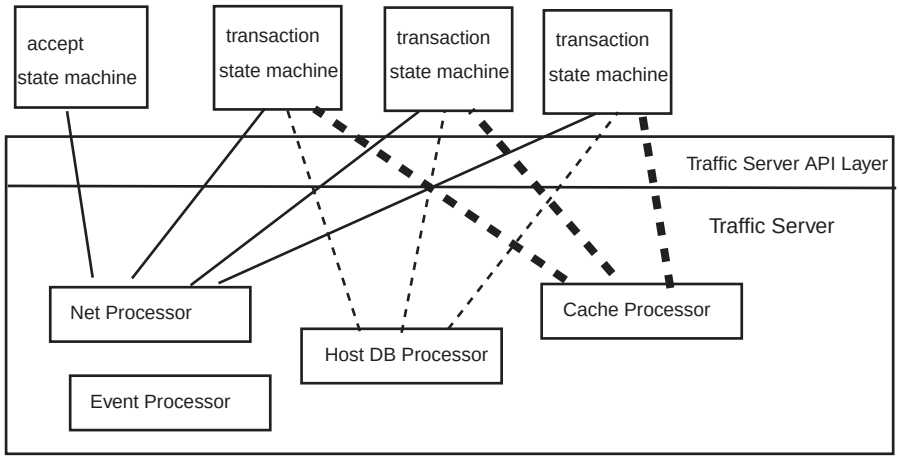


Figure 14 Protocol plugin flow of events

The flow of events is illustrated in Figure 14. The thin straight lines show Net Processor event flow, the thin dashed lines are Host DB event flow, and the thick dashed lines are Cache event flow.

Notice that this flow of events is independent of the design of the Protocol plugin (whether you build "accept" and "transaction" state machines or not). Any plugin that supports network connections uses the net vconnection interfaces (`INKNetAccept`, `INKNetConnect`) and thus receives events from Net Processor. Any plugin that performs cache lookups or cache writes uses `INKCacheRead`, `INKCacheWrite`, `INKVConnRead`, and `INKVConnWrite` and thus receives events from Cache Processor and the Traffic Edge event system; similarly, any plugin that does DNS lookups receives events from the Host DB Processor.

One way to implement a transaction state machine

The transaction state machines (TSMs) in the Protocol plugin have to do several things:

- Keep track of the state of the transaction
- Handle the events they receive (based on the state of the transaction and the event received)
- Update the state of the transaction as it changes

Here is one way you can implement TSMs (details on how the Protocol plugin does this follow in the next section):

- Create a data structure for transactions that contains all of the state data you need to keep track of. In the Protocol plugin this is a struct, `Txn_SM`.
- When you create the TSM's continuation, initialize data of type `Txn_SM`. Initialize the data to the initial state of a transaction (in this case, a net connection has just been accepted). Associate this data to the TSM continuation using `INKContDataSet`.
- Write state handler functions that handle the expected events for each state.
- Write the handler for the TSM. Its job is to receive events, examine the current state, and execute the appropriate state handler function. In the Protocol plugin, the handler is `main_handler`. `main_handler` calls the state handler functions to handle each state.

The flow of execution is illustrated in Figure 15.

1. The handler for the TSM, (called `main_handler` in the Protocol plugin) receives the TSM's events.
2. `main_handler` examines the state of the transaction—in particular, it examines the current handler.
3. `main_handler` calls the `current_handler`, which is one of the state handler functions, and passes `current_handler` the current event. In Figure 15, the current handler is `state2_handler`.
4. The `current_handler` handles the event, and updates the data. In Figure 15, the state is changed from `state2` to `state3` (and the current handler is changed from `state2_handler` to `state3_handler`). The next time main_handler receives an event, it will be processed by `state3_handler`.
5. `state2_handler` arranges the next callback of the TSM. Typically, it gives Traffic Edge additional work to do (such as writing a file to cache), in order to progress to the next state. The TSM (`main_handler`) then waits for the next event to arrive from Traffic Edge.

This implementation is diagrammed in Figure 15. The details are provided in the next section, a walk through the processing of a typical transaction.

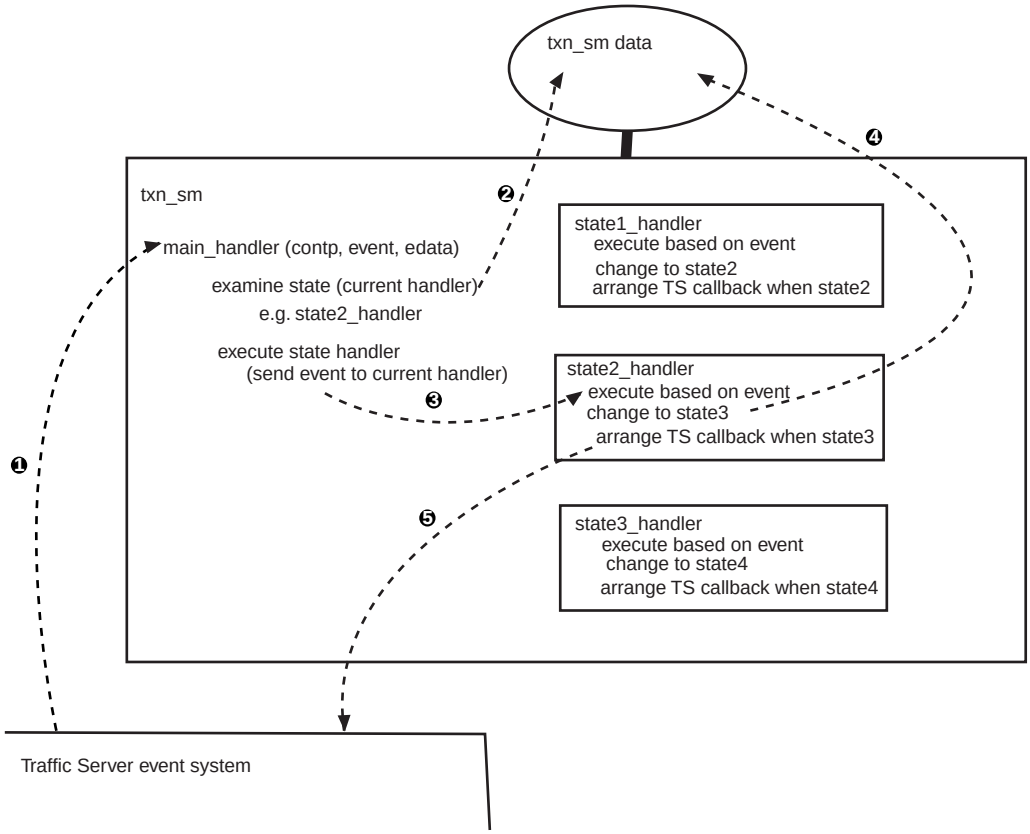


Figure 15 How transaction state machines are implemented in the Protocol plugin

Processing a typical transaction

The code is contained in the following files:

- `Protocol.c` and `Protocol.h`
- `Accept.c` and `Accept.h`
- `TxnSM.c` and `TxnSM.h`

Here is a step-by-step run-through of the code.

1. The `INKPluginInit` function is in `Protocol.c`. It checks the validity of the `plugin.config` entries (there must be two, a client accept port and a server port), and runs an initialization routine, `init`.
2. The `init` function (in `Protocol.c`) creates the plugin's log file using `INKTextLogObjectCreate`.
3. The `init` function creates the accept state machine using `AcceptCreate`. The code for `AcceptCreate` is in `Accept.c`.
4. The accept state machine, like the transaction state machine, keeps track of its state via a data structure. This data structure, `Accept`, is defined in `Accept.h`. In

AcceptCreate, state data is associated to the new accept state machine using INKContDataSet.

5 The init function arranges the callback of the accept state machine when there is a network connection using INKNetAccept.

6 The handler for the accept state machine is accept_event in Accept.c. When Traffic Edge's Net Processor sends INK_EVENT_NET_ACCEPT to the accept state machine, accept_event creates a transaction state machine, txn_sm, by calling TxnSMCreate. Notice that accept_event creates a mutex for the transaction state machine; each transaction state machine has its own mutex.

7 The TxnSMCreate function is in TxnSM.c. The first thing it does is to initialize the transaction's data. This data is of type TxnSM (defined in TxnSM.h). Notice that the current handler (q_current_handler) is set to state_start.

8 Then TxnSMCreate creates a transaction state machine using INKContCreate. The handler for the transaction state machine is main_handler.

9 main_handler is in TxnSM.c. When accept_event receives INK_EVENT_NET_ACCEPT, it calls the transaction state machine (INKContCall (txn_sm, 0, NULL);). The event passed to main_handler is 0 (INK_EVENT_NONE).

10 The first thing main_handler does is examine the current txn_sm state by calling INKContDataGet. The state is state_start.

11 main_handler invokes the handler for state_start by using the function pointer TxnSMHandler (defined in TxnSM.h).

12 The state_start handler function (in TxnSM.c) is handed an event (at this stage, the event is INK_EVENT_NET_ACCEPT) and a client vconnection. state_start checks to see if this client vconnection is closed; if not, state_start attempts to read data from the

client vconnection into an INKIOBuffer. (state_start is handling the event it receives).

13 state_start changes the current handler to state_interface_with_client. (Updates the state of the transaction to the next state).

14 state_start initiates a read of the client vconnection (arranges for Traffic Edge to send INK_EVENT_VCONN_READ_READY events to the TSM), by calling INKVConnRead.

15 state_interface_with_client is activated by the next event from Traffic Edge. It checks for errors, and examines the read VIO for the read operation initiated by INKVConnRead.

16 If the read VIO is the client_read_VIO (which we are expecting at this stage in the transaction), state_interface_with_client updates the state to state_read_request_from_client.

17 state_read_request_from_client handles actual INK_EVENT_READ_READY events and reads the client request.

18 state_read_request_from_client parses the client request.

19 state_read_request_from_client updates the state to the next state, state_handle_cache_lookup.

20 state_read_request_from_client arranges for Traffic Edge to call back the TSM with the next set of events , initiating the cache lookup, by calling INKCacheRead.

21 When the INKCacheRead sends the TSM INK_EVENT_OPEN_READ (a cache hit) or INK_EVENT_OPEN_READ_FAILED (a cache miss), main_handler calls state_handle_cache_lookup.

CHAPTER 6

HTTP Hooks and Transactions

Hooks are points in Traffic Edge transaction processing where plugins can step in and do some work. Registering a plugin function for callback amounts to "adding" the function to a hook. You can register your plugin to be called back for every single transaction, or for specific transactions only.

This chapter contains the following sections:

Transformation hooks are discussed in *"Transformations" on page 42*.

The set of hooks

First you need the following terminology

HTTP transaction

A transaction consists of a single HTTP request from a client and the response that Traffic Edge sends to that client. A transaction begins when Traffic Edge receives a request, and ends when Traffic Edge sends the response.

Traffic Edge uses HTTP state machines to process transactions. The state machines follow a complex set of states involved in sophisticated caching and document retrieval (taking into account, for example, alternate selection, freshness criteria, and hierarchical caching). The Traffic Edge API provides hooks to a subset of these states, illustrated in Figure 16, on page 66.

transform hooks

The two transform hooks, `INK_HTTP_REQUEST_TRANSFORM_HOOK` and `INK_HTTP_RESPONSE_TRANSFORM_HOOK` are called in the course of an HTTP transform. To see where in the HTTP transaction they are called, look for the "set up transform" ovals in Figure 16, on page 66.

HTTP session

A session consists of a single client connection to Traffic Edge. A session can consist of several transactions, in succession. The session starts when the client connection opens, and ends when the connection closes.

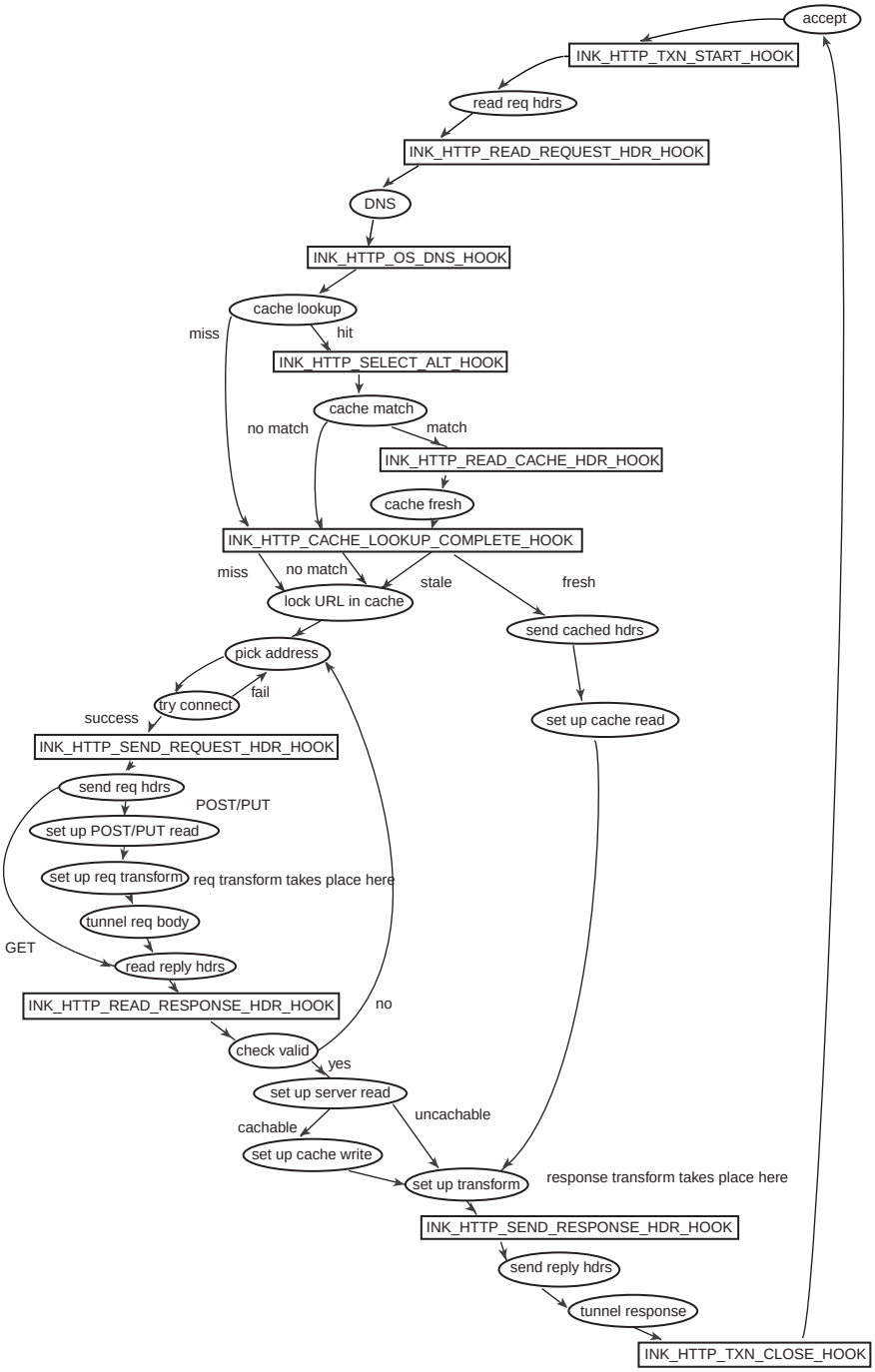


Figure 16 HTTP transaction state diagram

Adding hooks

There are several ways of adding hooks to your plugin.

global HTTP hooks

HTTP transaction hooks are set on a global basis using the function `INKHttpHookAdd`. This means that the continuation specified as the parameter to `INKHttpHookAdd` is called for every transaction. `INKHttpHookAdd` must be used in `INKPluginInit`.

transaction hooks

Transaction hooks can be used to call plugins back for a specific HTTP transaction. You cannot add transaction hooks in `INKPluginInit`; you first need a handle to a transaction. See *"Accessing the Transaction Being Processed" on page 33*.

transformation hooks

Transformation hooks are a special case of transaction hooks. See *"INKVConnCacheObjectSizeGet" on page 220* for more information on the transformation hooks. You add a transformation hook using `INKHttpTxnHookAdd`, described in *"HTTP transactions" on page 69*.

session hooks

An HTTP session starts when a client opens a connection to Traffic Edge and ends when the connection closes. A session can consist of several transactions. Session hooks allow you to hook your plugin to a particular point in every transaction within a specified session. See *"HTTP sessions" on page 68*. Session hooks are added in a manner similar to transaction hooks (you first need a handle to an HTTP session).

HTTP select alternate hook

Alternate selection hooks allow you to hook on to the alternate selection state. These hooks must be added globally, since Traffic Edge does not have a handle to a transaction or session when alternate selection is taking place. See *"HTTP alternate selection" on page 73* for information on the alternate selection mechanism.

All of the hook addition functions (*INKHttpHookAdd*, *IINKHttpSsnHookAdd*, *INKHttpSsnReenable*)take an `INKHttpHookID` identifying the hook to add on to and an `INKCont` which is the basic callback mechanism in Traffic Edge. A single `INKCont` can be added to any number of hooks at a given time.

An HTTP hook is identified by the enumerated type `INKHttpHookID`. The values for `INKHttpHookID` are:

Values for INKHttpHookID	Description
INK_HTTP_READ_REQUEST_HDR_HOOK	Called immediately after the request header is read from the client. Corresponds to the event `INK_EVENT_HTTP_READ_REQUEST_HDR`.
INK_HTTP_OS_DNS_HOOK	Called immediately after the HTTP state machine has completed a DNS lookup of the origin server. The HTTP state machine will know the origin server's IP address at this point which is useful for performing both authentication and blacklisting. Corresponds to the event `INK_EVENT_HTTP_OS_DNS`.
INK_HTTP_SEND_REQUEST_HDR_HOOK	Called immediately before the proxy's request header is sent to the origin server or the parent proxy. Notice that this hook will not be called if the document is being served from cache. This hook is usually used for modifying the proxy's request header before it is sent to the origin server or parent proxy.

Values for INKHttpHookID	Description
INK_HTTP_READ_CACHE_HDR_HOOK	Called immediately after the request and response header of a previously cached object is read from cache. Notice that this hook will only be called if the document is being served from cache. Corresponds to the event `INK_EVENT_HTTP_READ_CACHE_HDR`.
INK_HTTP_READ_RESPONSE_HDR_HOOK	Called immediately after the response header is read from the origin server or parent proxy. Corresponds to the event `INK_EVENT_HTTP_READ_RESPONSE_HDR`.
INK_HTTP_SEND_RESPONSE_HDR_HOOK	Called immediately before the proxy's response header is written to the client. This hook is usually used for modifying the response header. Corresponds to the event `INK_EVENT_HTTP_SEND_RESPONSE_HDR`.
INK_HTTP_REQUEST_TRANSFORM_HOOK	See *"Transformations" on page 42* for information on the transformation hooks.
INK_HTTP_RESPONSE_TRANSFORM_HOOK	See *"Transformations" on page 42* for information on the transformation hooks.
INK_HTTP_TXN_START_HOOK	Called when an HTTP transaction is started. A transaction starts when either a client connects to Traffic Edge and data is available on the connection or a previous client connection left open for keep alive has new data available.
INK_HTTP_TXN_CLOSE_HOOK	Called when an HTTP transaction ends.
INK_HTTP_SELECT_ALT_HOOK	See *"HTTP alternate selection" on page 73* for information on the alternate selection mechanism.
INK_HTTP_SSN_START_HOOK	Called when an HTTP session is started. A session starts when a client connects to Traffic Edge. You can only add this hook as a global hook.
INK_HTTP_SSN_CLOSE_HOOK	Called when an HTTP session ends. A session ends when the client connection is closed. You can only add this hook as a global hook.
INK_HTTP_CACHE_LOOKUP_COMPLETE_HOOK	Called once the HTTP state machine has commpleted the cache lookup for the document requested in the ongoing transaction. Register this hook either using either `INKHttpTxnHookAdd` or `INKHttpHookAdd`. Corresponds to the event `INK_EVENT_HTTP_CACHE_LOOKUP_COMPLETE`.

The function you use to add a global HTTP hook is *"INKHttpHookAdd" on page 151*.

HTTP sessions

An HTTP session is an object that is defined for the lifetime of a client's TCP session. The Traffic Edge API allows you to add a global hook to the start or end of an HTTP session,

and you can add session hooks that call back your plugin for every transaction within a given session.

When a client connects to Traffic Edge it opens up a TCP connection and sends one or more HTTP requests. An individual request and its response make up an HTTP transaction. The HTTP session begins when the client opens the connection, and ends when the connection closes.

The HTTP session hooks are:

INK_HTTP_SSN_START_HOOK	Called when an HTTP session is started. A session starts when a client connects to Traffic Edge. This hook must be added as a global hook.
INK_HTTP_SSN_CLOSE_HOOK	Called when an HTTP session ends. A session ends when the client connection is closed. This hook must be added as a global hook.

You use the session hooks to get a handle to a session (an `INKHttpSsn` object) and then if you want your plugin to be called back for each transaction within the session, you use `INKHttpSsnHookAdd`.

Note that you must reenable the session with `INKHttpSsnReenable` after processing a session hook.

The session hook functions are:

- *"IINKHttpSsnHookAdd" on page 152*
- *"INKHttpSsnReenable" on page 153*

HTTP transactions

The HTTP transaction functions allow you to set up plugin callbacks to HTTP transactions, and obtain and modify information about particular HTTP transactions.

As described in the section on HTTP sessions, an HTTP transaction is an object defined for the lifetime of a single request from a client and the response from Traffic Edge. The `INKHttpTxn` structure is the main handle given to a plugin for manipulating internal state about a transaction. Additionally, an HTTP transaction has a reference back to the HTTP session that created it.

Below is a sample of code that illustrates how to register locally to a transaction and associate data to the transaction.

```
/*
 * Simple plugin that illustrates:
 *  - how to register locally to a txn
 *  - how to deal with data associated to a txn
 *
 * Note: for code lisibility, error checking is omitted
 */
```

```
#include "InkAPI.h"

#define DBG_TAG "txn"


/* Structure to be associated to txns */
typedef struct {
  int i;
  float f;
  char *s;
} TxnData;


/* Allocate memory and init a TxnData structure */
TxnData *
txn_data_alloc()
{
  TxnData *data;
  data = INKmalloc(sizeof(TxnData));

  data->i = 1;
  data->f = 0.5;
  data->s = "Constant String";

  return data;
}

/* Free up a TxnData structure */
void
txn_data_free(TxnData *data)
{
  INKfree(data);
}

/* handler for event READ_REQUEST and TXN_CLOSE */
static int
local_hook_handler (INKCont contp, INKEvent event, void *edata)
{
  INKHttpTxn txnp = (INKHttpTxn) edata;
  TxnData *txn_data = INKContDataGet(contp);
```

```
  switch (event) {
  case INK_EVENT_HTTP_READ_REQUEST_HDR:
    /* Modify values of txn data */
    txn_data->i = 2;
    txn_data->f = 3.5;
    txn_data->s = "Constant String 2";
    break;

  case INK_EVENT_HTTP_TXN_CLOSE:
    /* Print txn data values */
   INKDebug(DBG_TAG, "Txn data i=%d f=%f s=%s", txn_data->i, txn_data->f,
txn_data->s);

    /* Then destroy the txn cont and it's data */
    txn_data_free(txn_data);
    INKContDestroy(contp);
    break;

  default:
     INKAssert(!"Unexpected event");
     break;
   }

   INKHttpTxnReenable(txnp, INK_EVENT_HTTP_CONTINUE);
   return 1;
}

/* Handler for event TXN_START */
static int
global_hook_handler (INKCont contp, INKEvent event, void *edata)
{
   INKHttpTxn txnp = (INKHttpTxn) edata;
   INKCont txn_contp;
   TxnData *txn_data;

   switch (event) {
   case INK_EVENT_HTTP_TXN_START:
     /* Create a new continuation for this txn and associate data to it */
     txn_contp = INKContCreate(local_hook_handler, INKMutexCreate());
     txn_data = txn_data_alloc();
     INKContDataSet(txn_contp, txn_data);

     /* Registers locally to hook READ_REQUEST and TXN_CLOSE */
```

```
      INKHttpTxnHookAdd(txnp, INK_HTTP_READ_REQUEST_HDR_HOOK, txn_contp);
      INKHttpTxnHookAdd(txnp, INK_HTTP_TXN_CLOSE_HOOK, txn_contp);
      break;

    default:
      INKAssert(!"Unexpected event");
      break;
    }

    INKHttpTxnReenable(txnp, INK_EVENT_HTTP_CONTINUE);
    return 1;
}



void
INKPluginInit (int argc, const char *argv[])
{
  INKCont contp;

  /* Note that we do not need a mutex for this txn as it registers globally
     and doesn't have any data associated with it */
  contp = INKContCreate(global_hook_handler, NULL);

  /* Register gloabally */
  INKHttpHookAdd(INK_HTTP_TXN_START_HOOK, contp);
}
```

See *"Adding hooks" on page 67* for background about HTTP transactions, and HTTP hooks. See Figure 16, on page 66, for an illustration of the steps involved in a typical HTTP transaction.

The HTTP transaction functions are:

- *"INKHttpTxnCacheLookupStatusGet" on page 154*
- *"INKHttpTxnCachedReqGet" on page 154*

  Note that it is an error to modify cached headers.
- *"INKHttpTxnCachedRespGet" on page 155*

  Note that it is an error to modify cached headers.
- *"INKHttpTxnClientIncomingPortGet" on page 155*
- *"INKHttpTxnClientIPGet" on page 155*
- *"INKHttpTxnClientRemotePortGet" on page 156*

- *“NKHttpTxnClientReqGet” on page 156*

  Plugins that must read client request headers use this call to retrieve the HTTP header.
- *“INKHttpTxnClientRespGet” on page 156*
- *“INKHttpTxnErrorBodySet” on page 157*
- *“INKHttpTxnHookAdd” on page 157*
- *“INKHttpTxnNextHopIPGet” on page 158*
- *“INKHttpTxnNextHopIPGet” on page 158*
- *“INKHttpTxnParentProxySet” on page 158*
- *“INKHttpTxnReenable” on page 159*
- *“INKHttpTxnServerIPGet” on page 159*
- *“INKHttpTxnServerReqGet” on page 160*
- *“INKHttpTxnServerRespGet” on page 160*
- *“INKHttpTxnSsnGet” on page 160*
- *“INKHttpTxnTransformedRespCache” on page 161*
- *“INKHttpTxnTransformRespGet” on page 161*
- *“INKHttpTxnUntransformedRespCache” on page 162*

Intercepting HTTP Transactions

The intercepting HTTP transaction functions provide plugins the ability to intercept transactions either after the request is received or on contact with the origin server. The plugin acts as the origin server using the INKVConn interface. Allows both for reading POST bodies in plugins as well as using alternative transports to the origin server.

The intercepting HTTP transaction functions are:

- *“INKHttpTxnIntercept” on page 163*
- *“INKHttpTxnServerIntercept” on page 164*

Initiate HTTP Connection

The initiate HTTP connection function allows plugins to initiate HTTP transactions. The initiate HTTP connection function is:

- *“INKHttpConnect” on page 162*

HTTP alternate selection

The HTTP alternate selection functions provide a mechanism for hooking into Traffic Edge’s alternate selection mechanism and augmenting it with additional information.

HTTP alternate selection refers to the process of choosing between several alternate versions of a document for a given URL. Alternates arise because the HTTP 1.1 specification allows different documents to be sent back for the same URL depending on the clients request. For example, a server might send back a GIF image to a client who only accepts GIF images and might send back a JPEG image to a client who only accepts JPEG images.

The alternate selection mechanism is invoked when Traffic Edge looks up a URL in its cache. For each URL Traffic Edge stores a vector of alternates. For each alternate in this vector, Traffic Edge computes a quality value between `0` and `1` for how "good" the alternate is. A quality value of `0` means that the alternate is unacceptable. A quality value of `1` means that the alternate is a perfect match.

If a plugin hooks onto the`INK_HTTP_SELECT_ALT_HOOK` it will be called back when Traffic Edge performs alternate selection. You cannot register locally to the hook `INK_HTTP_SELECT_ALT_HOOK` by using `INKHttpTxnHookAdd`, but by using only `INKHttpHookAdd`. It is only valid to hook onto the global list of `INK_HTTP_SELECT_ALT_HOOK`'s since Traffic Edge does not actually have an HTTP transaction or an HTTP session on hand when alternate selection is performed. Traffic Edge calls each of the select alternate hooks with the event `INK_EVENT_HTTP_SELECT_ALT.` The `void *`*`edata`* argument that is passed to the continuation is a pointer to an `INKHttpAltInfo` structure. It can be used later to call the HTTP alternate selection functions listed at the end of this section. Unlike other hooks, this alternate selection callout is non-blocking and the expectation is that the quality value for the alternate will be changed by a call to `INKHttpAltInfoQualitySet`.

Note HTTP SM does not have to be reenabled using `INKHttpTxnReenable` or any other APIs. Just return from the function.

Below is a sample of code that illustrates how to call the Alternate APIs.

```
static void handle_select_alt(INKHttpAltInfo infop)
{
    INKMBuffer client_req_buf, cache_resp_buf;
    INKMLoc client_req_hdr, cache_resp_hdr;

    INKMLoc accept_transform_field;
    INKMLoc content_transform_field;

    int accept_transform_len = -1, content_transform_len = -1;
    const char* accept_transform_value = NULL;
    const char* content_transform_value = NULL;
    int content_plugin, accept_plugin;

    float quality;

    /* get client request, cached request and cached response */
    INKHttpAltInfoClientReqGet (infop, &client_req_buf, &client_req_hdr);
    INKHttpAltInfoCachedRespGet(infop, &cache_resp_buf, &cache_resp_hdr);
```

```
    /* get the Accept-Transform field value from the client request */
    accept_transform_field = INKMimeHdrFieldFind(client_req_buf,
client_req_hdr, "Accept-Transform", -1);
    if (accept_transform_field) {
        INKMimeHdrFieldValueStringGet(client_req_buf, client_req_hdr,
accept_transform_field,
                                      0, &accept_transform_value,
&accept_transform_len);
        INKDebug(DBG_TAG, "Accept-Transform = |%s|",
accept_transform_value);
    }


    /* get the Content-Transform field value from cached server response
*/
    content_transform_field = INKMimeHdrFieldFind(cache_resp_buf,
cache_resp_hdr, "Content-Transform", -1);
    if (content_transform_field) {
        INKMimeHdrFieldValueStringGet(cache_resp_buf, cache_resp_hdr,
content_transform_field,
                                      0, &content_transform_value,
&content_transform_len);
        INKDebug(DBG_TAG, "Content-Transform = |%s|",
content_transform_value);
    }


    /* compute quality */
    accept_plugin = (accept_transform_value && (accept_transform_len > 0)
&&
                     (strncmp(accept_transform_value,  "plugin",
accept_transform_len) == 0));


    content_plugin = (content_transform_value && (content_transform_len >
0) &&
                      (strncmp(content_transform_value,  "plugin",
content_transform_len) == 0));


    if (accept_plugin) {
        quality = content_plugin ? 1.0 : 0.0;
    } else {
        quality = content_plugin ? 0.0 : 0.5;
    }


    INKDebug(DBG_TAG, "Setting quality to %3.1f", quality);


    /* set quality for this alternate */
    INKHttpAltInfoQualitySet(infop, quality);
```

```
    /* cleanup */
    if (accept_transform_value)
        INKHandleStringRelease(client_req_buf, accept_transform_field,
accept_transform_value);
    if (accept_transform_field)
        INKHandleMLocRelease(client_req_buf, client_req_hdr,
accept_transform_field);
    INKHandleMLocRelease(client_req_buf, INK_NULL_MLOC, client_req_hdr);


    if (content_transform_value)
        INKHandleStringRelease(cache_resp_buf, content_transform_field,
content_transform_value);
    if (content_transform_field)
        INKHandleMLocRelease(cache_resp_buf, cache_resp_hdr,
content_transform_field);
    INKHandleMLocRelease(cache_resp_buf, INK_NULL_MLOC, cache_resp_hdr);
}




static int alt_plugin(INKCont contp, INKEvent event, void *edata)
{
    INKHttpAltInfo infop;

    switch (event) {
    case INK_EVENT_HTTP_SELECT_ALT:
        infop = (INKHttpAltInfo)edata;
        handle_select_alt(infop);
        break;

    default:
        break;
    }

    return 0;
}

void INKPluginInit (int argc, const char *argv[])
{
    INKHttpHookAdd(INK_HTTP_SELECT_ALT_HOOK, INKContCreate (alt_plugin,
NULL));
```

```
}
```

Traffic Edge augments the alternate selection through these callouts using the following algorithm.

1. Traffic Edge computes its own quality value for the alternate. Traffic Edge takes into account the quality of the accept match, the encoding match and the language match.
2. Traffic Edge then calls out each of the continuations on the global `INK_HTTP_SELECT_ALT_HOOK`'s list.
3. It multiplies its quality value with the value returned by each callout. Since all of the values are clamped to be between `0` and `1`, the final value will be between `0` and `1`.
4. This algorithm also ensures that a single callout can block the usage of a given alternate by specifying a quality value of `0`.

A common usage for the alternate selection mechanism is when a plugin transforms a document for some clients and not for others and wants to store both the transformed and un-transformed document. The client's request would specify whether it accepted the transformed document and the plugin could then determine if the alternate matched this specification and set the quality level for the alternate appropriately.

The HTTP alternate selection functions are:

- *"INKHttpAltInfoCachedReqGet" on page 165*
- *"INKHttpAltInfoCachedRespGet" on page 166*
- *"INKHttpAltInfoClientReqGet" on page 166*
- *"INKHttpAltInfoQualitySet" on page 166*

CHAPTER 7 Miscellaneous Interface Guide

Most of the functions in the Traffic Edge API provide an interface to specific code modules within Traffic Edge. The miscellaneous functions described in this chapter provide some useful general capabilities:

While the C library already provides functions such as `printf`, `malloc`, and `fopen` that perform these tasks, the Traffic Edge API versions overcome various C library limitations (such as portability to all Traffic Edge-supported platforms).

Debugging functions

The debugging functions give you the following debugging capabilities:

- *"INKDebug" on page 143* prints out a formatted statement if you are running Traffic Edge in debug mode.
- *"INKIsDebugTagSet" on page 144* finds out if a debug tag is set. If the debug tag is set, Traffic Edge prints out any debug statements associated to the debug tag.
- *"INKError" on page 144* prints error messages to Traffic Edge's error log.
- *"INKAssert" on page 144* allows the use of assertion in a plugin.
- *"INKReleaseAssert" on page 145* allows the use of assertion in a plugin.

The INKfopen family

The `fopen` family of functions in C is normally used for reading configuration files, since `fgets` is an easy way to parse files on a line by line basis. The `INKfopen` family of functions is aimed at solving the same problem of buffered IO and line at a time IO in a platform independent manner. The `INKfopen` family of functions works exactly the same under Microsoft Windows NT as it does under any of the Unix platforms Traffic Edge runs on. Further, the `fopen` family of C library functions can only open a file if a file descriptor less than 256 is available. Traffic Edge often has more than 2000 file descriptors open at once, making the likelihood of an available file descriptor less than 256 very small. The `INKfopen` family can open files with descriptors greater than 256.

INKfopen not optimized for speed

The INKfopen family of routines is not intended for high speed IO or for flexibility, but are blocking APIs, not asynchronous. Thus, for performance reasons, it is recommended not to directly use these APIs on a TS thread (when being called back on an HTTP hook). It is better to use a separate thread for doing the blocking IO. The INKfopen family is intended for reading and writing configuration information when corresponding usage of the fopen family of functions is inappropriate because of file descriptor and portability limitations. The INKfopen family of functions consists of:

- *"INKfclose" on page 146*
- *"INKfflush" on page 146*
- *"INKfgets" on page 146*
- *"INKfopen" on page 146*
- *"INKfread" on page 147*
- *"INKfwrite" on page 148*

Memory allocation

Traffic Edge provides five routines for allocating and freeing memory. These routines correspond to similar routines in the C library. For example, INKrealloc behaves like the C library routine realloc. There are two reasons to use the routines provided by Traffic Edge. The first is portability. The Traffic Edge API routines behave the same on all of Traffic Edge's supported platforms. For example, realloc does not accept an argument of NULL on some platforms. The second reason is that the Traffic Edge routines actually track the memory allocations by file and line number. This tracking is very efficient, is always turned on, and is useful for tracking down memory leaks.

The memory allocation functions are:

- *"INKfree" on page 148*
- *"INKmalloc" on page 148*
- *"INKrealloc" on page 149*
- *"INKstrdup" on page 149*
- *"INKstrndup" on page 149*

Thread functions

The Traffic Edge API thread functions enable you to create, destroy, and identify threads within Traffic Edge. Multithreading enables a single program to have more than one stream of execution and to process more than one transaction at a time.

Threads serialize their access to shared resources and data using the INKMutex type, described in *"Mutexes" on page 101*.

The thread functions are:

CHAPTER 8

HTTP Headers

This chapter is about the functions used to manipulate HTTP headers.

About HTTP headers

An HTTP message consists of:

- An HTTP header
- body
- trailer

The HTTP header consists of:

- Request or response line
  - An HTTP request line is composed of a method, a URL and version
  - A response line is composed of a version, a status code and a reason phrase
- MIME header

A MIME header is made up of zero or more MIME fields. A MIME field is composed of a field name, a colon and zero or more field values. The values in a field are separated by commas.

An HTTP header containing a request line is usually referred to as a request. The following example shows a typical request header.

Example request

```
GET http://www.inktomi.com/ HTTP/1.0
Proxy-Connection: Keep-Alive
User-Agent: Mozilla/4.08 [en] (X11; I; Linux 2.2.3 i686)
Host: www.inktomi.com
Accept: image/gif, image/x-xbitmap, image/jpeg, image/pjpeg, image/png, */
*
Accept-Encoding: gzip
Accept-Language: en
Accept-Charset: iso-8859-1, *, utf-8
```

The response header for the above request might look like the following:

Example response

```
HTTP/1.0 200 OK
Date: Mon, 29 Mar 1999 06:57:43 GMT
Content-Location: http://locutus.inktomi.com/index.html
Etag: "07db14afa76be1:1074"
Last-Modified: Thu, 25 Mar 1999 20:01:38 GMT
Content-Length: 7931
Content-Type: text/html
Server: Microsoft-IIS/4.0
Age: 922
Proxy-Connection: close
```

The following figure illustrates an HTTP message, with the HTTP header blown up:

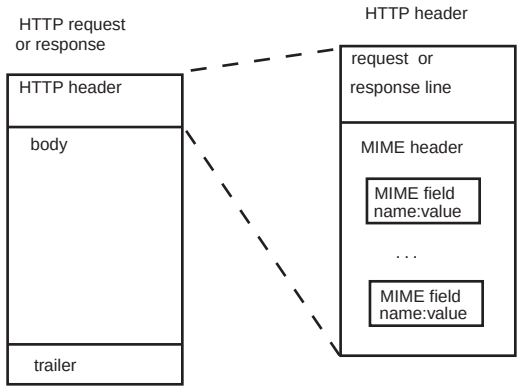


Figure 17 HTTP request/response and header structure

The following figure gives examples of HTTP request and response headers.

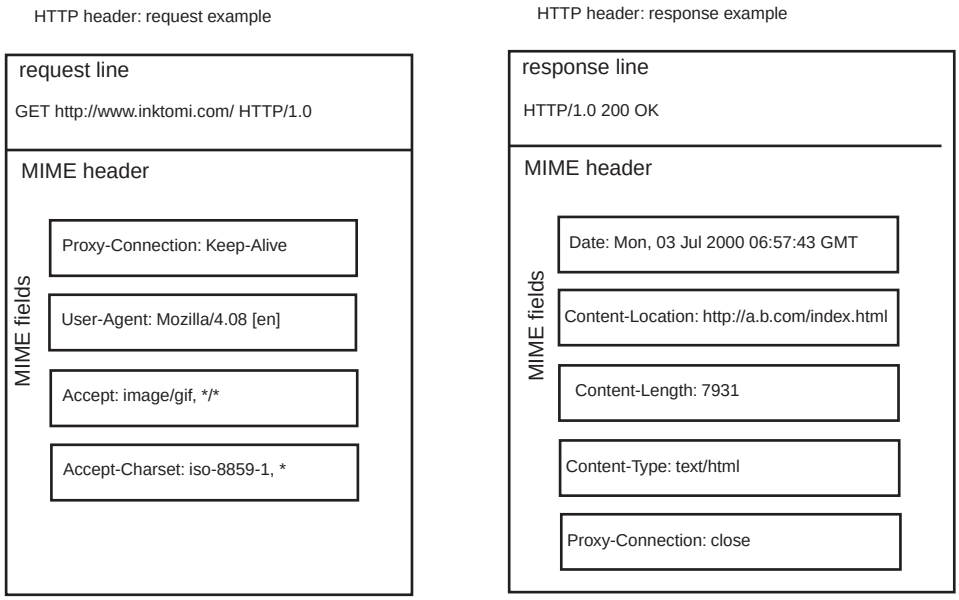


Figure 18 Examples of HTTP request and response headers

accessing HTTP header data

The marshal buffer or `INKMBuffer` is a heap data structure that stores parsed URLs, MIME headers and HTTP headers. You can allocate new objects out of marshal buffers, and change the values within the marshal buffer. Whenever you manipulate an object, you require the handle to the object (`INKMLoc`) and the marshal buffer containing the object (`INKMBuffer`).

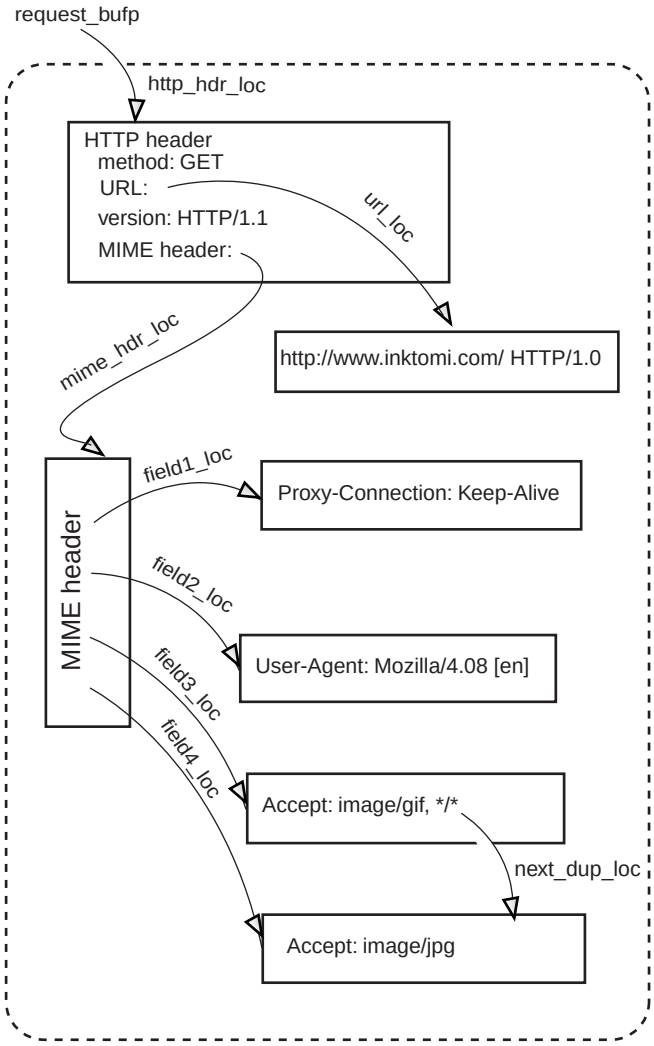


Figure 19 Marshal buffers and header locations

Figure 19 shows:

- The marshal buffer containing the HTTP request, `reqest_bufp`
- `INKMLoc` location pointer for the HTTP header (`http_hdr_loc`)
- `INKMLoc` location pointer for the request URL (`url_loc`)
- `INKMLoc` location pointers for the MIME header (`mime_hdr_loc`)
- `INKMLoc` location pointers for MIME fields (`field`*`i`*`_loc`)
- `INKMLoc` location pointer for the next duplicate MIME field (`next_dup_loc`)

The diagram also shows that an HTTP header contains pointers to the URL location and the MIME header location. You can obtain the URL location from an HTTP header using the function `INKHttpHdrUrlGet`. To work with MIME headers, you can pass either a MIME header location or an HTTP header location to MIME header functions . If you pass

an HTTP header to a MIME header function, the system locates the associated MIME header and executes the MIME header function on the MIME header location.

Guide to Traffic Edge HTTP header system

Please read this section.

IMPORTANT Previous versions of Traffic Edge are named Traffic Server. Throughout this manual, Traffic Server, Traffic Server 3.0, Traffic Server 3.5, and Traffic Server 5.2 refer to previous versions of Traffic Edge. For version checking, Traffic Edge 1.5 is equivalent to Traffic Server 5.5.

Older (pre-4.0) versions of Traffic Server's header processing system analysed and disassembled HTTP headers for convenience, at considerable performance cost. New performance enhancements do not assume this breakdown and reassembly. The consequences are the following.

No null-terminated strings

In Traffic Server 5.2 and newer, you cannot assume that the string data contained in marshal buffers (data such as URLs and MIME fields) is stored in null-terminated string copies. This means that your plugins should always use the length parameter when retrieving or manipulating these strings. You **cannot** pass in `NULL` for string-length return values. String values returned from marshall buffers are not null-terminated. If you need a null-terminated value, use `INKstrndup` to automatically null-terminate a string. The strings that come back, which are not null-terminated, **cannot** be passed into the common `str*()` routines.

Note Values returned from a marshall buffer can be `NULL`, which means the field or object requested does not exist.

For example (from the `blacklist-1` sample):

```
char *host_string;
int host_length;
host_string = INKUrlHostGet (bufp, url_loc, &host_length);
for (i = 0; i < nsites; i++) {
    if (strncmp (host_string, sites[i], host_length) == 0) {
        ...
    }
```

See the sample plugins for more examples.

Duplicate MIME fields are not coalesced

MIME headers may contain more than one MIME field with the same name. Pre-4.0 versions of Traffic Server joined multiple fields with the same name into one field with composite values. This behavior comes at a performance cost, and causes interoperability problems with some older clients and servers. Traffic Server 4.0 and newer ceases coalescing duplicate fields.

Correctly behaving plugins should check for the presence of duplicate fields, and iterate over the duplicate fields, by using `INKMimeHdrFieldNextDup` (see *"INKMimeHdrFieldNextDup" on page 191*).

MIME fields always belong to an associated MIME header

In Traffic Server versions 4.0 and newer, you cannot create a new MIME field without an associated MIME header or HTTP header; MIME fields are always seen as part of a MIME header or HTTP header.

To use a MIME field, you must specify the MIME header or HTTP header to which it belongs. This header is called the field's *parent header*. The `INKMimeField*` functions in pre-2.0 versions of the SDK, which do not require the parent header as inputs, have been deprecated. SDK 2.0 has new functions, the `INKMimeHdrField*` series, that require you to specify the location of the parent header along with the location of the MIME field. For every deprecated `INKMimeField*` function, there is a new preferred `INKMimeHdrField*` function. Use the `INKMimeHdrField*` functions instead of the deprecated `INKMimeField*` series. Here are some examples:

Instead of:

INKMLoc **`INKMimeFieldCreate`**`(INKMBuffer` *`bufp`*`)`

Use:

INKMLoc **`INKMimeHdrFieldCreate`**`(INKMBuffer` *`bufp`*`, INKMLoc` *`hdr`*`)`

Instead of:

```
void INKMimeFieldCopyValues (INKMBuffer dest_bufp, INKMLoc dest_offset,
   INKMBuffer src_bufp, INKMLoc src_offset)
```

Use:

```
void INKMimeHdrFieldCopyValues (INKMBuffer dest_bufp, INKMLoc dest_hdr,
   INKMLoc dest_field, INKMBuffer src_bufp, INKMLoc src_hdr, INKMLoc
   src_field)
```

In the `INKMimeHdrField*` function prototypes, the `INKMLoc field` corresponds to the `INKMLoc offset` used the `INKMimeField*` functions. See the discussion of parent `INKMLoc` in the following section.

Release marshal buffer handles

When you fetch a component object or create a new object, you get back a handle to the object location. The handle is either an `INKMLoc` for an object location, or a char * for a string location. You can manipulate the object through these handles, but when you are finished, you need to release the handle to free up system resources.

The general guideline is to release all `INKMLoc` and string handles you retrieve. The one exception is the string returned by `INKUrlStringGet`, which must be freed by a call to `INKfree`.

the parent location

The handle release functions expect three arguments: the marshal buffer containing the data, the location of the parent object, and the location of the object to be released. The parent location is usually clear from the creation of the `INKMLoc` or string; for example, if your plugin had the following calls:

```
url_loc = INKHttpHdrUrlGet (bufp, hdr_loc);
```

```
host_string = INKUrlHostGet (bufp, url_loc, &host_length);
```

Your plugin would have to call:

```
INKHandleStringRelease (bufp, url_loc, host_string);
INKHandleMLocRelease (bufp, hdr_loc, url_loc);
```

null parent

If an `INKMLoc` is obtained from a transaction, it does not have a parent `INKMLoc`. Use the null `INKMLoc` constant `INK_NULL_MLOC` as its parent. For example, if your plugin calls:

```
INKHttpTxnClientReqGet (txnp, &bufp, &hdr_loc);
```

You must release `hdr_loc` with:

```
INKHandleMLocRelease (bufp, INK_NULL_MLOC, hdr_loc);
```

when to use null parent

You need to use `INK_NULL_MLOC` to release any `INKMLoc` handles retrieved by the `INKHttpTxn*Get` functions.

Here's an example using a new `INKMimeHdrField` function:

```
INKHttpTxnServerRespGet( txnp, &resp_bufp, &resp_hdr_loc );
new_field_loc = INKMimeHdrFieldCreate (resp_bufp, resp_hdr_loc);
INKHandleMLocRelease ( resp_bufp, resp_hdr_loc, new_field_loc);
INKHandleMLocRelease ( resp_bufp, INK_NULL_MLOC, resp_hdr_loc);
```

See the sample plugins for many more examples.

Tip

Release handles before reenabling the HTTP transaction. In other words, call `INKHandleMLocRelease` or `INKHandleStringRelease` before `INKHttpTxnReenable`. See the sample code.

Deprecated functions

Several marshal buffer functions and MIME field functions are deprecated in this release. The following marshal buffer functions are deprecated. Do *not* use them:

- `INKMBufferCompress`
- `INKMBufferDataGet`
- `INKMBufferDataSet`
- `INKMBufferLengthGet`
- `INKMBufferRef`
- `INKMBufferUnref`

The following MIME field functions are deprecated. If you need to support these functions in existing code, documentation is provided in *"Deprecated Functions" on page 253*.

- `INKMimeFieldCreate`
- `INKMimeFieldDestroy`
- `INKMimeFieldCopy`
- `INKMimeFieldCopyValues`

- INKMimeFieldNext
- INKMimeFieldLengthGet
- INKMimeFieldNameGet
- INKMimeFieldNameSet
- INKMimeFieldValuesClear
- INKMimeFieldValuesCount
- INKMimeFieldValueGet
- INKMimeFieldValueGetInt
- INKMimeFieldValueGetUint
- INKMimeFieldValueGetDate
- INKMimeFieldValueSet
- INKMimeFieldValueSetInt
- INKMimeFieldValueSetUint
- INKMimeFieldValueSetDate
- INKMimeFieldValueAppend
- INKMimeFieldValueInsert
- INKMimeFieldValueInsertInt
- INKMimeFieldValueInsertUint
- INKMimeFieldValueInsertDate
- INKMimeFieldValueDelete
- INKMimeHdrFieldValueGet
- INKMimeHdrFieldValueGetDate
- INKMimeHdrFieldValueGetInt
- INKMimeHdrFieldValueGetUint
- INKMimeHdrFieldValueInsert
- INKMimeHdrFieldValueInsertDate
- INKMimeHdrFieldValueInsertInt
- INKMimeHdrFieldValueInsertUint
- INKMimeHdrFieldValueSet
- INKMimeHdrFieldValueSetDate
- INKMimeHdrFieldValueSetInt
- INKMimeHdrFieldValueSetUint
- INKMimeHdrFieldDelete
- INKMimeHdrFieldInsert
- INKMimeHdrFieldRetrieve

Marshal buffers

The marshal buffer or `INKMBuffer` is a heap data structure that stores parsed URLs, MIME headers and HTTP headers. You can allocate new objects out of marshal buffers, and change the values within the marshal buffer. Whenever you manipulate an object, you require the handle to the object (`INKMLoc`) and the marshal buffer containing the object (`INKMBuffer`).

Routines exist for manipulating the object based on these two pieces of information. See, for example:

- *"HTTP headers" on page 91*
- *"URLs" on page 94*
- *"MIME headers" on page 95*

The marshal buffer functions allow you to create and destroy Traffic Edge's marshal buffers, which are the data structures that hold parsed URLs, MIME headers, and HTTP headers.

Caution

Any marshal buffer fetched by `INKHttpTxn*Get` will be used by other parts of the system. Be careful not to destroy these shared, transaction marshal buffers. In functions such as:

`INKHttpTxnClientReqGet`

`INKHttpTxnClientRespGet`

`INKHttpTxnServerReqGet`

`INKHttpTxnServerRespGet`

`INKHttpTxnCachedReqGet`

`INKHttpTxnCachedRespGet`

`INKHttpTxnTransformRespGet`

the parameters `INKMBuffer`, `bufp`, `INKMLoc` and `loc` are output parameters and the buffer `bufp` should not be a created `MBuffer`. Also, the handle to the header (`loc`) should be released using the `INKHandleMLocRelease` function. Lastly, the `MBuffer` returned by the above functions should not be destroyed by the user.

The marshal buffer-specific functions are:

- `INKMBufferCreate`
- `INKMBufferDestroy`

HTTP headers

The Traffic Edge API HTTP header functions enable you to work with HTTP header data stored in marshal buffers.

HTTP header data structure

The HTTP header data structure is a parsed version of the HTTP header defined in the HTTP protocol specification. An HTTP header is composed of a request or response line followed by zero or more MIME fields. In fact, an HTTP header is a subclass of a MIME header and all of the MIME header routines operate on HTTP headers.

An HTTP request line is composed of a method, a URL and version. A response line is composed of a version, a status code and a reason phrase. See *"About HTTP headers" on page 83* for details and examples of HTTP headers.

In order to facilitate fast comparisons and to reduce storage size, Traffic Edge defines several pre-allocated method names. These names correspond to the methods defined in the HTTP 1.1 specification.

Pre-allocated method names	HTTP 1.1 method
INK_HTTP_METHOD_CONNECT	"CONNECT"
INK_HTTP_METHOD_DELETE	"DELETE"
INK_HTTP_METHOD_GET	"GET"
INK_HTTP_METHOD_HEAD	"HEAD"
INK_HTTP_METHOD_ICP_QUERY	"ICP_QUERY"
INK_HTTP_METHOD_OPTIONS	"OPTIONS"
INK_HTTP_METHOD_POST	"POST"
INK_HTTP_METHOD_PURGE	"PURGE"
INK_HTTP_METHOD_PUT	"PUT"
INK_HTTP_METHOD_TRACE	"TRACE"

Traffic Edge also defines several common values that appear in HTTP headers.

Traffic Edge definition	HTTP header value
INK_HTTP_VALUE_BYTES	"bytes"
INK_HTTP_VALUE_CHUNKED	"chunked"
INK_HTTP_VALUE_CLOSE	"close"
INK_HTTP_VALUE_COMPRESS	"compress"
INK_HTTP_VALUE_DEFLATE	"deflate"
INK_HTTP_VALUE_GZIP	"gzip"
INK_HTTP_VALUE_IDENTITY	"identity"
INK_HTTP_VALUE_KEEP_ALIVE	"keep-alive"
INK_HTTP_VALUE_MAX_AGE	"max-age"
INK_HTTP_VALUE_MAX_STALE	"max-stale"
INK_HTTP_VALUE_MIN_FRESH	"min-fresh"
INK_HTTP_VALUE_MUST_REVALIDATE	"must-revalidate"
INK_HTTP_VALUE_NONE	"none"
INK_HTTP_VALUE_NO_CACHE	"no-cache"
INK_HTTP_VALUE_NO_STORE	"no-store"
INK_HTTP_VALUE_NO_TRANSFORM	"no-transform"
INK_HTTP_VALUE_ONLY_IF_CACHED	"only-if-cached"
INK_HTTP_VALUE_PRIVATE	"private"
INK_HTTP_VALUE_PROXY_REVALIDATE	"proxy-revalidate"

Traffic Edge definition	HTTP header value
INK_HTTP_VALUE_PUBLIC	"public"
INK_HTTP_VALUE_S_MAX_AGE	"s-maxage"

The method names and header values above are defined in `InkAPI.h` as `const char*` strings. When Traffic Edge sets a method or a header value it makes a quick check to see if the new value is one of the known values. If it is, instead of storing the known value in the marshal buffer it stores a pointer into a global table. The method names and header values listed above are also pointers into this table. This allows simple pointer comparison of the value returned from `INKHttpMethodGet` with one of the values listed above. It is also recommended that you use the above values when referring to one of the known schemes as doing so removes the possibility of a spelling error.

The HTTP header functions are:

- `INKHttpHdrClone`
- `INKHttpHdrCopy`
- `INKHttpHdrCreate`
- `INKHttpHdrDestroy`
- `INKHttpHdrLengthGet`
- `INKHttpHdrMethodGet`
- `INKHttpHdrMethodSet`
- `INKHttpHdrPrint`
- `INKHttpHdrReasonGet`
- `INKHttpHdrReasonLookup`
- `INKHttpHdrReasonSet`
- `INKHttpHdrStatusGet`
- `INKHttpHdrStatusSet`
- `INKHttpHdrTypeGet`
- `INKHttpHdrTypeSet`
- `INKHttpHdrUrlGet`
- `INKHttpHdrUrlSet`
- `INKHttpHdrVersionGet`
- `INKHttpHdrVersionSet`
- `INKHttpParserClear`
- `INKHttpParserCreate`
- `INKHttpParserDestroy`
- `INKHttpHdrParseReq`
- `INKHttpHdrParseResp`

URLs

The URL data structure is a parsed version of a standard internet URL. The Traffic Edge API URL functions provide access to URL data stored in marshal buffers. The URL functions can create, copy, retrieve or delete entire URLs, and retrieve or modify parts of URLs, such as their port or scheme information.

URL structure

The general form of an Internet URL is:

```
scheme://user:password@host:port/stuff
```

The URL data structure includes support for two specific types of internet URLs. HTTP URLs have the form:

```
http://user:password@host:port/path;params?query#fragment
```

FTP URLs have the form:

```
ftp://user:password@host:port/path;type=val
```

URL data storage

The URL port and FTP type are stored as integers. All remaining parts of the URL (the scheme, user, etc.) are stored as strings.

URL function naming

URL functions are named according to the portion of the URL on which they operate. For instance, the function that retrieves the `host` portion of a URL is named `INKUrlHostGet`.

To facilitate fast comparisons and to reduce storage size, Traffic Edge defines several pre-allocated scheme names.

Traffic Edge definition	Pre-allocated scheme name	URL scheme string lengths
INK_URL_SCHEME_FILE	“file”	INK_URL_LEN_FILE
INK_URL_SCHEME_FTP	“ftp”	INK_URL_LEN_FTP
INK_URL_SCHEME_GOPHER	“gopher”	INK_URL_LEN_GOPHER
INK_URL_SCHEME_HTTP	“http”	INK_URL_LEN_HTTP
INK_URL_SCHEME_HTTPS	“https”	INK_URL_LEN_HTTPS
INK_URL_SCHEME_MAILTO	“mailto”	INK_URL_LEN_MAILTO
INK_URL_SCHEME_NEWS	“news”	INK_URL_LEN_NEWS
INK_URL_SCHEME_NNTP	“nntp”	INK_URL_LEN_NNTP
INK_URL_SCHEME_PROSPERO	“prospero”	INK_URL_LEN_PROSPERO
INK_URL_SCHEME_TELNET	“telnet”	INK_URL_LEN_TELNET
INK_URL_SCHEME_WAIS	“wais”	INK_URL_LEN_WAIS

The scheme names above are defined in `InkAPI.h` as `const char*` strings. When Traffic Edge sets the scheme portion of the URL (or any portion for that matter), it makes a quick check to see if the new value is one of the known values. If it is, instead of storing the known value in the marshal buffer, it stores a pointer into a global table. The scheme values listed above are also pointers into this table. This allows simple pointer comparison of the value returned from `INKUrlSchemeGet` with one of the values listed above. Inktomi recommends that you use the Traffic Edge-defined values when referring to one of the known schemes, as doing so removes the possibility of a spelling error.

The URL functions are:

- `INKUrlClone`
- `INKUrlCopy`
- `INKUrlCreate`
- `INKUrlDestroy`
- `INKUrlPrint`
- `INKUrlFtpTypeGet`
- `INKUrlFtpTypeSet`
- `INKUrlHostGet`
- `INKUrlHostSet`
- `INKUrlHttpFragmentGet`
- `INKUrlHttpFragmentSet`
- `INKUrlHttpParamsGet`
- `INKUrlHttpParamsSet`
- `INKUrlHttpQueryGet`
- `INKUrlHttpQuerySet`
- `INKUrlLengthGet`
- `INKUrlParse`
- `INKUrlPasswordGet`
- `INKUrlPasswordSet`
- `INKUrlPathGet`
- `INKUrlPathSet`
- `INKUrlPortGet`
- `INKUrlPortSet`
- `INKUrlSchemeGet`
- `INKUrlSchemeSet`
- `INKUrlStringGet`
- `INKUrlUserGet`
- `INKUrlUserSet`

MIME headers

The Traffic Edge API MIME header functions enable you to retrieve and modify information about HTTP MIME fields.

An HTTP request or response consists of a header, body, and trailer. The HTTP header consists of a request or response line, and a MIME header. A MIME header is composed of

zero or more MIME fields. A MIME field is composed of a field name, a colon and zero or more field values. The values in a field are separated by commas. In the following example, `Foo` is the MIME field name and `bar` is the first MIME field value and `car` is the second MIME field value:

Example

```
Foo: bar, car
```

The following is an augmented Backus-Naur Form (BNF) for the form of a MIME header. It specifies exactly what was described above. A header consists of zero or more fields which consist of a name, a separating colon and zero or more values. A name or value is simply a string of tokens which is potentially zero length. And a token is any character except certain control characters and separators such as colons.

Example

```
MIME-header = *MIME-field

MIME-field = field-name ":" #field-value

field-name = *token

field-value = *token
```

For the purposes of retrieving a field, field names are not case sensitive: the field names `Foo`, `foo` and `fOO` are all equivalent.

The MIME header data structure is a parsed version of a standard Internet MIME header. The MIME header data structure is similar to the URL data structure (see *"URLs" on page 94*). The actual data is stored in a marshal buffer and the MIME header functions operate on a marshal buffer and a location (`INKMLoc`) within the buffer.

After a call to `INKMimeHdrFieldDestroy`, `INKMimeHdrFieldRemove` or `INKUrlDestroy` is made, you must deallocate the `INKMLoc` handle by a call to `INKHandleMLocRelease`. You do not need to deallocate a `NULL` handles. For instance, if you called `INKMimeHdrFieldValueStringGet` to get the value of the content type field and the field does not exist, it returns `INK_NULL_MLOC`. In this case, you would not have to deallocate the handle by a call to `INKHandleMLocRelease`.

MIME header locations

The location (`INKMLoc`) in the following MIME header functions can be either a HTTP header location or a MIME header location. If an HTTP header location is passed to these function, the system locates the MIME header associated with this HTTP header, and executes the corresponding MIME header operations specified by the functions. See the example in the description of *"INKMimeHdrCopy" on page 198*.

MIME headers may contain more than one MIME field with the same name. Previous versions of Traffic Edge (Traffic Server versions before 4.0) joined multiple fields with the same name into one field with composite values. This behavior comes at a performance cost, and causes interoperability problems with some older clients and servers. Future versions of Traffic Edge will cease coalescing duplicate fields.

Correctly behaving plugins should check for the presence of duplicate fields, and iterate over the duplicate fields, by using `INKMimeHdrFieldNextDup`.

To facilitate fast comparisons and to reduce storage size, Traffic Edge defines several pre-allocated field names. These field names correspond to field names found in HTTP and NNTP headers.

Traffic Edge pre-allocated field names	HTTP and NNTP header field names	Associated string lengths
INK_MIME_FIELD_ACCEPT	"Accept"	INK_MIME_LEN_ACCEPT
INK_MIME_FIELD_ACCEPT_CHARSET	“Accept-Charset"	INK_MIME_LEN_ACCEPT_CHARSET
INK_MIME_FIELD_ACCEPT_ENCODING	"Accept-Encoding"	INK_MIME_LEN_ACCEPT_ENCODING
INK_MIME_FIELD_ACCEPT_LANGUAGE	"Accept-Language"	INK_MIME_LEN_ACCEPT_LANGUAGE
INK_MIME_FIELD_ACCEPT_RANGES	"Accept-Ranges"	INK_MIME_LEN_ACCEPT_RANGES
INK_MIME_FIELD_AGE	"Age"	INK_MIME_LEN_AGE
INK_MIME_FIELD_ALLOW	"Allow"	INK_MIME_LEN_ALLOW
INK_MIME_FIELD_APPROVED	“Approved"	INK_MIME_LEN_APPROVED
INK_MIME_FIELD_AUTHORIZATION	"Authorization"	INK_MIME_LEN_AUTHORIZATION
INK_MIME_FIELD_BYTES	"Bytes"	INK_MIME_LEN_BYTES
INK_MIME_FIELD_CACHE_CONTROL	"Cache-Control"	INK_MIME_LEN_CACHE_CONTROL
INK_MIME_FIELD_CLIENT_IP	"Client-ip"	INK_MIME_LEN_CLIENT_IP
INK_MIME_FIELD_CONNECTION	"Connection"	INK_MIME_LEN_CONNECTION
INK_MIME_FIELD_CONTENT_BASE	"Content-Base"	INK_MIME_LEN_CONTENT_BASE
INK_MIME_FIELD_CONTENT_ENCODING	"Content-Encoding"	INK_MIME_LEN_CONTENT_ENCODING
INK_MIME_FIELD_CONTENT_LANGUAGE	"Content-Language"	INK_MIME_LEN_CONTENT_LANGUAGE
INK_MIME_FIELD_CONTENT_LENGTH	"Content-Length"	INK_MIME_LEN_CONTENT_LENGTH
INK_MIME_FIELD_CONTENT_LOCATION	"Content-Location"	INK_MIME_LEN_CONTENT_LOCATION
INK_MIME_FIELD_CONTENT_MD5	"Content-MD5"	INK_MIME_LEN_CONTENT_MD5
INK_MIME_FIELD_CONTENT_RANGE	"Content-Range"	INK_MIME_LEN_CONTENT_RANGE
INK_MIME_FIELD_CONTENT_TYPE	"Content-Type"	INK_MIME_LEN_CONTENT_TYPE
INK_MIME_FIELD_CONTROL	"Control"	INK_MIME_LEN_CONTROL
INK_MIME_FIELD_COOKIE	"Cookie"	INK_MIME_LEN_COOKIE
INK_MIME_FIELD_DATE	"Date"	INK_MIME_LEN_DATE
INK_MIME_FIELD_DISTRIBUTION	"Distribution"	INK_MIME_LEN_DISTRIBUTION
INK_MIME_FIELD_ETAG	"Etag"	INK_MIME_LEN_ETAG
INK_MIME_FIELD_EXPECT	"Expect"	INK_MIME_LEN_EXPECT
INK_MIME_FIELD_EXPIRES	"Expires"	INK_MIME_LEN_EXPIRES
INK_MIME_FIELD_FOLLOWUP_TO	"Followup-To"	INK_MIME_LEN_FOLLOWUP_TO
INK_MIME_FIELD_FROM	"From"	INK_MIME_LEN_FROM
INK_MIME_FIELD_HOST	"Host"	INK_MIME_LEN_HOST
INK_MIME_FIELD_IF_MATCH	"If-Match"	INK_MIME_LEN_IF_MATCH

Traffic Edge pre-allocated field names	HTTP and NNTP header field names	Associated string lengths
INK_MIME_FIELD_IF_MODIFIED_SINCE	"If-Modified-Since"	INK_MIME_LEN_IF_MODIFIED_SINCE
INK_MIME_FIELD_IF_NONE_MATCH	"If-None-Match"	INK_MIME_LEN_IF_NONE_MATCH
INK_MIME_FIELD_IF_RANGE	"If-Range"	INK_MIME_LEN_IF_RANGE
INK_MIME_FIELD_IF_UNMODIFIED_SINCE	"If-Unmodified-Since"	INK_MIME_LEN_IF_UNMODIFIED_SINCE
INK_MIME_FIELD_KEEP_ALIVE	"Keep-Alive"	INK_MIME_LEN_KEEP_ALIVE
INK_MIME_FIELD_KEYWORDS	"Keywords"	INK_MIME_LEN_KEYWORDS
INK_MIME_FIELD_LAST_MODIFIED	"Last-Modified"	INK_MIME_LEN_LAST_MODIFIED
INK_MIME_FIELD_LINES	"Lines"	INK_MIME_LEN_LINES
INK_MIME_FIELD_LOCATION	"Location"	INK_MIME_LEN_LOCATION
INK_MIME_FIELD_MAX_FORWARDS	"Max-Forwards"	INK_MIME_LEN_MAX_FORWARDS
INK_MIME_FIELD_MESSAGE_ID	"Message-ID"	INK_MIME_LEN_MESSAGE_ID
INK_MIME_FIELD_NEWSGROUPS	"Newsgroups"	INK_MIME_LEN_NEWSGROUPS
INK_MIME_FIELD_ORGANIZATION	"Organization"	INK_MIME_LEN_ORGANIZATION
INK_MIME_FIELD_PATH	"Path"	INK_MIME_LEN_PATH
INK_MIME_FIELD_PRAGMA	"Pragma"	INK_MIME_LEN_PRAGMA
INK_MIME_FIELD_PROXY_AUTHENTICATE	"Proxy-Authenticate"	INK_MIME_LEN_PROXY_AUTHENTICATE
INK_MIME_FIELD_PROXY_AUTHORIZATION	"Proxy-Authorization"	INK_MIME_LEN_PROXY_AUTHORIZATION
INK_MIME_FIELD_PROXY_CONNECTION	"Proxy-Connection"	INK_MIME_LEN_PROXY_CONNECTION
INK_MIME_FIELD_PUBLIC	"Public"	INK_MIME_LEN_PUBLIC
INK_MIME_FIELD_RANGE	"Range"	INK_MIME_LEN_RANGE
INK_MIME_FIELD_REFERENCES	"References"	INK_MIME_LEN_REFERENCES
INK_MIME_FIELD_REFERER	"Referer"	INK_MIME_LEN_REFERER
INK_MIME_FIELD_REPLY_TO	"Reply-To"	INK_MIME_LEN_REPLY_TO
INK_MIME_FIELD_RETRY_AFTER	"Retry-After"	INK_MIME_LEN_RETRY_AFTER
INK_MIME_FIELD_SENDER	"Sender"	INK_MIME_LEN_SENDER
INK_MIME_FIELD_SERVER	"Server"	INK_MIME_LEN_SERVER
INK_MIME_FIELD_SET_COOKIE	"Set-Cookie"	INK_MIME_LEN_SET_COOKIE
INK_MIME_FIELD_SUBJECT	"Subject"	INK_MIME_LEN_SUBJECT
INK_MIME_FIELD_SUMMARY	"Summary"	INK_MIME_LEN_SUMMARY
INK_MIME_FIELD_TE	"TE"	INK_MIME_LEN_TE
INK_MIME_FIELD_TRANSFER_ENCODING	"Transfer-Encoding"	INK_MIME_LEN_TRANSFER_ENCODING
INK_MIME_FIELD_UPGRADE	"Upgrade"	INK_MIME_LEN_UPGRADE
INK_MIME_FIELD_USER_AGENT	"User-Agent"	INK_MIME_LEN_USER_AGENT
INK_MIME_FIELD_VARY	"Vary"	INK_MIME_LEN_VARY

Traffic Edge pre-allocated field names	HTTP and NNTP header field names	Associated string lengths
INK_MIME_FIELD_VIA	"Via"	INK_MIME_LEN_VIA
INK_MIME_FIELD_WARNING	"Warning"	INK_MIME_LEN_WARNING
INK_MIME_FIELD_WWW_AUTHENTICATE	"Www-Authenticate"	INK_MIME_LEN_WWW_AUTHENTICATE
INK_MIME_FIELD_XREF	"Xref"	INK_MIME_LEN_XREF

The header field names above are defined in `InkAPI.h` as `const char*` strings. When Traffic Edge sets the name portion of a header field (or any portion for that matter) it makes a quick check to see if the new value is one of the known values. If it is, instead of storing the known value in the marshal buffer it stores a pointer into a global table. The header field names listed above are also pointers into this table. This allows simple pointer comparison of the value returned from `INKMimeHdrFieldNameGet` with one of the values listed above. It is also recommended that you use the above values when referring to one of the known header field names as doing so removes the possibility of a spelling error.

custom MIME fields

Traffic Edge adds one important feature to MIME fields that those people already familiar with MIME headers will not know about. Namely, Traffic Edge does not print a MIME field if the field name begins with the `'@'` symbol. For example, a plugin can add the field `"@My-Field"` to a header. Even though Traffic Edge never sends that field out in a request to an origin server or in a response to a client, they can be printed in TS logs by defining a custom log config file that explicitly logs these fields. This provides a useful mechanism for plugins to store information about an object in one of the MIME headers associated with the object.

The MIME header functions are:

- `INKMimeHdrFieldClone`
- `INKMimeHdrFieldCopy`
- `INKMimeHdrFieldCopyValues`
- `INKMimeHdrFieldCreate`
- `INKMimeHdrFieldDestroy`
- `INKMimeHdrFieldLengthGet`
- `INKMimeHdrFieldNameGet`
- `INKMimeHdrFieldNameSet`
- `INKMimeHdrFieldNext`
- `INKMimeHdrFieldNextDup`
- `INKMimeHdrFieldValueAppend`
- `INKMimeHdrFieldValueDelete`
- `INKMimeHdrFieldValuesClear`
- `INKMimeHdrFieldValuesCount`
- `INKMimeHdrClone`
- `INKMimeHdrCopy`
- `INKMimeHdrCreate`

- INKMimeHdrDestroy
- INKMimeHdrFieldFind
- INKMimeHdrFieldGet
- INKMimeHdrFieldRemove
- INKMimeHdrFieldsClear
- INKMimeHdrFieldsCount
- INKMimeHdrLengthGet
- INKMimeHdrParse
- INKMimeParserClear
- INKMimeParserCreate
- INKMimeParserDestroy
- INKMimeHdrPrint

CHAPTER 9

Mutex Guide

Use mutexes to lock shared data. This chapter explains how to use the mutex interface.

Mutexes

A mutex is the basic synchronization method used within Traffic Edge to protect data from simultaneous access by multiple threads. A mutex acts as a lock that protects data in one program thread from being accessed by another thread.

Important: use TryLock when possible

The Traffic Edge API provides two functions that attempt to access and lock the data: `InkMutexLockTry` and `INKMutexLock`. `INKMutexLock` is a blocking call; if you use it, you can slow Traffic Edge performance (transaction processing pauses until the mutex is unlocked). It should be used only on threads created by the plugin (`INKContThreadCreate`). Never use it on a continuation handler called back by HTTP SM or Cache, Net or Event Processor. Even if the critical section is very small, do not use it. If you need to update a flag, set a variable, use atomic operations. If `INKMutexLock` is used in any case other than the one recommended above, the result will cause serious performance impact. `INKMutexLockTry`, on the other hand, attempts to lock the mutex only if it is unlocked (not being used by another thread). It should be used in all cases other than the above mentioned `INKMutexLock` case. If the `INKMutexLockTry` attempt fails, you can schedule a future attempt, which must be at least 10 milliseconds later. See for an example.

Inktomi recommends that, in general, you use `INKMutexLockTry` rather than `INKMutexLock`.

- `InkMutexLockTry` is *required* if you are tying to lock Traffic Edge internal or system resources, such as network, cache, eventProcessor, HTTP state machines and IO buffers.
- `InkMutexLockTry` is *required* if you are making any blocking calls, such as network or cache or file IO calls.
- `INKMutexLock` might not be necessary if you are not making blocking calls, and if you are only accessing local resources.

Traffic Edge API uses the `INKMutex` type for a mutex.

2 typical ways to use mutexes

There are two typical uses of mutex. One use is to lock global data or data shared by various continuations. The other typical usage is to lock data associated to a continuation (data that might be accessed by other continuations).

Locking global data

The `blacklist-1.c` sample plugin implements an example of this type. The blacklist plugin reads its blacklisted sites from a configuration file. File read operations are protected by a mutex created in `INKPluginInit`. The `blacklist-1.c` code uses

INKMutexLockTry instead of InkMutexLock. See *"blacklist-1.c" on page 245* for the blacklist-1.c code (start by looking at the INKPluginInit function). The general guideline for locking shared data is:

1. Create a mutex for this shared data using INKMutexCreate.
2. Whenever you need to read or modify this data, first lock it by calling InkMutexLockTry. Then read or modify the data.
3. When you are done with the data, unlock it with INKMutexUnlock. If you are unlocking data accessed during the processing of an HTTP transaction, you must unlock it before calling INKHttpTxnReenable.

Protecting a continuation's data

You need to create a mutex to protect a continuation's data if it might be accessed by other continuations or processes.

To protect the data associated to a continuation, follow these steps:

1. Create a mutex for the continuation using INKMutexCreate. For example,

```
INKMutex mutexp;

mutexp = INKMutexCreate ();
```

2. When you create the continuation, specify this mutex as the continuation's mutex. For example,

```
INKCont contp;

contp = INKContCreate (handler, mutexp);
```

If any other functions want to access contp's data, it is up to them to get contp's mutex (using, for example, INKContMutexGet) and lock it. See the sample Protocol plugin for usage.

How to associate a continuation to every HTTP transaction

There might be several reasons to create a continuation for each HTTP transaction that calls back your plugin. Some examples include:

- register hooks locally with the new continuation instead of registering them globally to the continuation plugin.
- store data specific to each HTTP transaction that you might need to reuse across various hooks.
- use of APIs (like INKHostLookup) which will call back this continuation with a certain event.

How to add the new continuation

A typical way of adding the new continuation is to register the plugin continuation to be called back by HTTP transactions globally when they reach INK_HTTP_TXN_START_HOOK. Refer to the example below using a transaction specific continuation called *txn_contp*.

```
void INKPluginInit(int argc, const char *argv[])
  {
```

```
        /* Plugin continuation */
        INKCont contp;
        if ((contp = INKContCreate (plugin_cont_handler, NULL)) ==
INK_ERROR_PTR) {
            LOG_ERROR("INKContCreate");
        } else {
   if (INKHttpHookAdd (INK_HTTP_TXN_START_HOOK, contp) == INK_ERROR) {
LOG_ERROR("INKHttpHookAdd");
}
        }
    }
```

In the plugin continuation handler, create teh new continuation *txn_contp*, and register it to be called back at INK_HTTP_TXN_CLOSE_HOOK:

```
static int plugin_cont_handler(INKCont contp, INKEvent event, void *edata)
    {
       INKHttpTxn txnp = (INKHttpTxn)edata;
      INKCont txn_contp;

        switch (event) {
        case INK_EVENT_HTTP_TXN_START:
            /* Create the HTTP txn continuation */
            txn_contp = INKContCreate(txn_cont_handler, NULL);

            /* Register txn_contp to be called back when txnp reaches
INK_HTTP_TXN_CLOSE_HOOK */
            if (INKHttpTxnHookAdd (txnp, INK_HTTP_TXN_CLOSE_HOOK,
txn_contp) == INK_ERROR) {
                LOG_ERROR("INKHttpTxnHookAdd");
            }

            break;

        default:
            INKAssert(!"Unexpected Event");
            break;
        }

        if (INKHttpTxnReenable(txnp, INK_EVENT_HTTP_CONTINUE) ==
INK_ERROR) {
            LOG_ERROR("INKHttpTxnReenable");
        }

        return 0;
    }
```

Have the *txn_contp* handler destory itself when the HTTP transaction is closed. If you forget, your plugin will have a big memory leak.

```
static int txn_cont_handler(INKCont txn_contp, INKEvent event, void
*edata)
     {
         INKHttpTxn txnp;

         switch (event) {
         case INK_EVENT_HTTP_TXN_CLOSE:
             txnp = (INKHttpTxn) edata;
             INKContDestroy(txn_contp);
             break;

         default:
             INKAssert(!"Unexpected Event");
             break;
         }

         if (INKHttpTxnReenable(txnp, INK_EVENT_HTTP_CONTINUE) ==
INK_ERROR) {
             LOG_ERROR("INKHttpTxnReenable");
         }

         return 0;
     }
```

How to store data specific to each HTTP transaction

For the example above, store the data in the *txn_contp* data structure. This means that you will create your own data structure. Suppose you want to store the state of the HTTP transaction:

```
   typedef struct {
         int state;
     } ContData;
```

You would need to allocate the memory and initialize this structure for each HTTP txnp. You can do that in the plugin continuation handler when it is called back with INK_EVENT_HTTP_TXN_START:

```
static int plugin_cont_handler(INKCont contp, INKEvent event, void *edata)
     {
        INKHttpTxn txnp = (INKHttpTxn)edata;
       INKCont txn_contp;
        ContData *contData;

         switch (event) {
```

```
        case INK_EVENT_HTTP_TXN_START:
            /* Create the HTTP txn continuation */
            txn_contp = INKContCreate(txn_cont_handler, NULL);

            /* Allocate and initialize the txn_contp data */
            contData = (ContData*) INKmalloc(sizeof(ContData));
            contData->state = 0;
            if (INKContDataSet(txn_contp, contData) == INK_ERROR) {
                LOG_ERROR("INKContDataSet");
            }

            /* Register txn_contp to be called back when txnp reaches
INK_HTTP_TXN_CLOSE_HOOK */
            if (INKHttpTxnHookAdd (txnp, INK_HTTP_TXN_CLOSE_HOOK,
txn_contp) == INK_ERROR) {
                LOG_ERROR("INKHttpTxnHookAdd");
            }

            break;

        default:
            INKAssert(!"Unexpected Event");
            break;
        }

        if (INKHttpTxnReenable(txnp, INK_EVENT_HTTP_CONTINUE) ==
INK_ERROR) {
            LOG_ERROR("INKHttpTxnReenable");
        }

        return 0;
    }
```

For accessing this data from anywhere, use `INKContDataGet`:

```
INKCont txn_contp;
    ContData *contData;

    contData = INKContDataGet(txn_contp);
    if (contData == INK_ERROR_PTR) {
        LOG_ERROR("INKContDataGet");
    }
    contData->state = 1;
```

Remember to free this memory before destroying the continuation:

```
static int txn_cont_handler(INKCont txn_contp, INKEvent event, void
*edata)
     {
         INKHttpTxn txnp;
         ContData *contData;


         switch (event) {
         case INK_EVENT_HTTP_TXN_CLOSE:
             txnp = (INKHttpTxn) edata;
             contData = INKContDataGet(txn_contp);
             if (contData == INK_ERROR_PTR) {
                 LOG_ERROR("INKContDataGet");
             } else {
                 INKfree(contData);
             }
             INKContDestroy(txn_contp);
             break;

         default:
             INKAssert(!"Unexpected Event");
             break;
         }

         if (INKHttpTxnReenable(txnp, INK_EVENT_HTTP_CONTINUE) ==
INK_ERROR) {
             LOG_ERROR("INKHttpTxnReenable");
         }

         return 0;
     }
```

Using locks

You do not need to use locks when a continuation has registered itself to be called back by HTTP hooks and it only uses the HTTP APIs. In the example above, the continuation *txn_contp* has registered itself to be called back at HTTP hooks, and it only uses the HTTP APIs. In this case only, it is safe to access data shared between *txnp* and *txn_contp* without grabbing a lock. In the example above *txn_contp* is created with a NULL mutex. This works because the HTTP transaction *txnp* is the only which will call back *txn_contp*, and you are guaranteed that *txn_contp* will be called back only one hook at a time. After processing is done *txn_contp* will reenable *txnp*.

In all other cases, you should create a mutex with the continuation. Basically in the case where you are using iocore APIs, or any other API where *txn_contp* is scheduled to be called back by a processor (the cache processor, the DNS processor...), a lock is needed.

This ensures that *txn_contp* will be called back only one at a time, (i.e. you are sure that *txn_contp* will not be called back by both *txnp* and by the cache processor simultaneously, which would result in a situation where you are executing two pieces of code in conflict!)

Special case: continuations created for HTTP transactions

continuations created in HTTP transactions do not need mutexes

If your plugin creates a new continuation for each HTTP transaction, you probably do not have to create a new mutex for it, because each HTTP transaction (`INKHttpTxn` object) already has its own mutex.

For example, if you have code such as the following, it is not necessary to specify a mutex for the continuation created in `txn_handler`:

```
static void
txn_handler (INKHttpTxn txnp, INKCont contp) {
   INKCont newCont;
   ....
   newCont = INKContCreate (newCont_handler, NULL);
   //It's not necessary to create a new mutex for newCont.

   ...

   INKHttpTxnReenable (txnp, INK_EVENT_HTTP_CONTINUE);
}

static int
test_plugin (INKCont contp, INKEvent event, void *edata) {
   INKHttpTxn txnp = (INKHttpTxn) edata;

   switch (event) {
   case INK_EVENT_HTTP_READ_REQUEST_HDR:
      txn_handler (txnp, contp);
      return 0;
   default:
      break;
   }
   return 0;
}
```

The mutex functions are:

- *"INKMutexCreate" on page 203*
- *"INKMutexLock" on page 204*
- *"INKMutexLockTry" on page 204*

CHAPTER 10

Continuations

The continuation interface is Traffic Edge's basic callback mechanism. Continuations are instances of the opaque data type `INKCont`. In its basic form a continuation represents a handler function and a mutex. This chapter contains:

Mutexes and data

A continuation must be created with a mutex if your continuation does one of the following:

- is registered globally (`INKHttpHookAdd` or `INKHttpSsnHookAdd`) to an HTTP hook and uses `INKContDataSet/Get`.
- is registered locally (`INKHttpTxnHookAdd`) but for multiple transactions and uses `INK ContDataSet/Get`.
- uses `INKCacheXXX`, `INKNetXXX`, `INKHostLookup` or `INKContSchedule` APIs.

Before being activated, a caller must grab the continuation's mutex. This requirement makes it possible for a continuation's handler function to safely access its data and to prevent it from being run by multiple callers at the same time. See the sample Protocol plugin for usage. The data protected by the mutex is: any global or continuation data associated to the continuation by `INKContDataSet`. This does not include the local data created by the continuation handler function. A typical example of continuations created with associated data structures and mutexes is the transaction state machine created in the sample Protocol plugin. See *"One way to implement a transaction state machine" on page 60*.

Reentrant Calls

A reentrant call occurs when the continuation passed as an argument to the API can be called in the same stack trace as the function calling the API. For instance, if you call `INKCacheRead` (*`contp, mykey`*), it is possible that *`contp`*'s handler will be called directly and then `INKCacheRead` returns. Caveats that could cause a possible issues if:

- a continuation has data associated with it (`INKContDataGet`).
- the reentrant call passes itself as a continuation to the reentrant API. In this case, the continuation should not try to access its data after having called the reentrant API. The reason for this is that data may be modified by the section of code of the continuation's handler that handles the event sent by the API. It is recommended that you always return after a reentrant call to avoid accessing something that has been deallocated.

Below is an example with an explaination.

```
continuation_handler (INKCont contp, INKEvent event, void *edata) {
    switch (event) {
    case event1:
```

```
        INKReentrantCall (contp);
        /* Return right away after this call */
        break;
    case event2:
        INKContDestroy (contp);
        break;
    }
}
```

The above example first assumes that the continuation is called back with *event1* and does the first reentrant call which schedules the continuation to receive *event2*. Because the call is reentrant, the processor calls back the continuation right away with *event2* and the continuation is destroyed. If you try to access the continuation, or one of its members after the reentrant call, you might access something that has been deallocated. To avoid accessing something that has been deallocated, never access the continuation or any of its members after a reentrant call, just exit the handler.

Note that most HTTP transaction plugin continuations do not need non-null mutexes, because they are called within the processing of an HTTP transaction and thus have the transaction's mutex.

null mutexes

It is also possible to specify a continuation's mutex as `NULL`. This should be done only when registering a continuation to a global hook, by a call to `INKHttpHookAdd`. In this case, the continuation can be called simultaneously by different instances of HTTP SM running on different threads. Having a mutex here would slow down Traffic Edge performance since all the threads will try to lock the same mutex. The drawback of not having a mutex is that such a continuation cannot have data associated with it (`INKContDataGet/Set` can not be used).

When using a `NULL` mutex, it is dangerous to access the continuation's data, but it is usually the case that continuations with `NULL` mutexes have no data associated with them. An example of such a continuation would be one that gets called back every time an HTTP request is read and determines from the request alone whether to let the request through or whether to reject it. An HTTP transaction gives its continuation data to the `contp`.

How to activate continuations

Continuations are activated when they receive an event or by `INKContSchedule`, which schedules a continuation to receive an event. They might receive an event because:

- Your plugin calls `INKContCall`
- The Traffic Edge HTTP state machine sends an event corresponding to a particular HTTP hook
- A Traffic Edge IO processor (such as cache processor or net processor) is letting a continuation know that there is (cache or network) data available to read or write. These callbacks are a result of using functions such `INKVConnRead/Write`, or `INKCacheRead/Write`

Writing handler functions

The handler function is the meat of the continuation. It is supposed to examine the event and event data and do something appropriate. The probable action might be to schedule another event for the continuation to received, or to open up a connection to a server or to destroy itself.

The continuation's handler function is a function of type `INKEventFunc`. Its arguments are a continuation, an event, and a pointer to some data (this data is passed to the continuation by the caller; do not confuse this data with the continuation's own data, associated by `INKContDataSet`). When the continuation is called back, the continuation and an event are passed to the handler function. The continuation is a handle to the same continuation that is invoked. The handler function typically has a switch statement to handle the events it receives:

```
static int some_handler (INKcont contp, INKEvent event, void *edata)
{
     .....
   switch(event) {
      case INK_EVENT_SOME_EVENT_1:
         do_some_thing_1;
         return;
      case INK_EVENT_SOME_EVENT_2:
         do_some_thing_2;
         return;
      case INK_EVENT_SOME_EVENT_3:
         do_some_thing_3;
         return;
      default: break;
      }
   return 0;
}
```

Caution

You might notice that a continuation cannot determine if more events are "in flight" towards it. Do not use `INKContDestroy` to delete a continuation before making sure that all incoming events, such as those sent because of `INKHttpTxnHookAdd`, have been handled.

*Events and void * data*

The following table lists events and the corresponding type of `void *` data passed to handler functions:

Event	Hook or API function that sends the event	void * data type
INK_EVENT_HTTP_READ_REQUEST_HDR	INK_HTTP_READ_REQUEST_HDR_HOOK	INKHttpTxn
INK_EVENT_HTTP_OS_DNS	INK_HTTP_OS_DNS_HOOK	INKHttpTxn

Event	Hook or API function that sends the event	void * data type
INK_EVENT_HTTP_SEND_REQUEST_HDR	INK_HTTP_SEND_REQUEST_HDR_HOOK	INKHttpTxn
INK_EVENT_HTTP_READ_CACHE_HDR	INK_HTTP_READ_CACHE_HDR_HOOK	INKHttpTxn
INK_EVENT_HTTP_READ_RESPONSE_HDR	INK_HTTP_READ_RESPONSE_HDR_HOOK	INKHttpTxn
INK_EVENT_HTTP_SEND_RESPONSE_HDR	INK_HTTP_SEND_RESPONSE_HDR_HOOK	INKHttpTxn
INK_EVENT_HTTP_SELECT_ALT	INK_HTTP_SELECT_ALT_HOOK	INKHttpTxn
INK_EVENT_HTTP_TXN_START	INK_HTTP_TXN_START_HOOK	INKHttpTxn
INK_EVENT_HTTP_TXN_CLOSE	INK_HTTP_TXN_CLOSE_HOOK	INKHttpTxn
INK_EVENT_HTTP_SSN_START	INK_HTTP_SSN_START_HOOK	INKHttpSsn
INK_EVENT_HTTP_SSN_CLOSE	INK_HTTP_SSN_CLOSE_HOOK	INKHttpSsn
INK_EVENT_NONE		
INK_EVENT_CACHE_LOOKUP_COMPLETE	INK_HTTP_CACHE_LOOKUP_COMPLETE_HOOK	INKHttpTxn
INK_EVENT_IMMEDIATE	INKVConnClose, INKVIOReenable, INKContSchedule	
INK_EVENT_IMMEDIATE	INK_HTTP_REQUEST_TRANSFORM_HOOK	
INK_EVENT_IMMEDIATE	INK_HTTP_RESPONSE_TRANSFORM_HOOK	
INK_EVENT_CACHE_OPEN_READ	INKCacheRead	Cache VC
INK_EVENT_CACHE_OPEN_READ_FAILED	INKCacheRead	Error code, see INK_CACHE_ERROR_XXX
INK_EVENT_CACHE_OPEN_WRITE	INKCacheWrite	Cache VC
INK_EVENT_CACHE_OPEN_WRITE_FAILED	INKCacheWrite	Error code, see INK_CACHE_ERROR_XXX
INK_EVENT _CACHE_REMOVE	INKCacheRemove	Nothing
INK_EVENT_CACHE_REMOVE_FAILED	INKCacheRemove	Error code, see INK_CACHE_ERROR_XXX
INK_EVENT_NET_ACCEPT	INKNetAccept, INKHttpTxnServerIntercept, INKHttpTxnIntercept	Net VConnection
INK_EVENT_NET_ACCEPT_FAILED	INKNetAccept, INKHttpTxnServerIntercept, INKHttpTxnIntercept	Nothing
INK_EVENT_HOST_LOOKUP	INKHostLookup	Null pointer - error Non null pointer - INKHostLookup Result
INK_EVENT_TIMEOUT	INKContSchedule	
INK_EVENT_ERROR		

Event	Hook or API function that sends the event	void * data type
INK_EVENT_VCONN_READ_READY	INKVConnRead	INKVConn
INK_EVENT_VCONN_WRITE_READY	INKVConnWrite	INKVConn
INK_EVENT_VCONN_READ_COMPLETE	INKVConnRead	INKVConn
INK_EVENT_VCONN_WRITE_COMPLETE	INKVConnWrite	INKVConn
INK_EVENT_VCONN_EOS	INKVConnRead	INKVConn
INK_EVENT_NET_CONNECT	INKNetConnect	INKVConn
INK_EVENT_NET_CONNECT_FAILED	INKNetConnect	INKVConn
INK_EVENT_HTTP_CONTINUE		
INK_EVENT_HTTP_ERROR		
INK_EVENT_MGMT_UPDATE	INKMgmtUpdateRegister	NULL

The continuation functions are:

- INKContCall
- INKContCreate
- INKContDataGet
- INKContDataSet
- INKContDestroy
- INKContMutexGet
- INKContSchedule

CHAPTER 11

Plugin Configurations

This chapter contains:

Plugin configurations

The `INKConfig` family of functions provides a mechanism for accessing and changing global configuration information within a plugin.

external web interface

If you want to set up a web interface for configuring your plugin through Traffic Manager, see *"Setting up a plugin management interface" on page 131*.

not Traffic Edge configuration

The functions discussed in this section do not examine or modify Traffic Edge configuration variables. To examine Traffic Edge configuration and statistics variables, see *"Reading Traffic Edge settings and statistics" on page 132*.

The `INKConfig` family of functions is designed to provide a fast and efficient mechanism for accessing and changing global configuration information within a plugin. Such a mechanism is simple enough to provide in a single-threaded program, but the translation to a multi-threaded program such as Traffic Edge is difficult. A common technique is to have a single mutex protect the global configuration information. The problem with this solution is that a single mutex becomes a performance bottleneck very quickly.

The `INKConfig` family of functions define an interface to storing and retrieving an opaque data pointer. Internally, Traffic Edge maintains reference count information about the data pointer so that a call to `INKConfigSet` will not disturb another thread using the current data pointer. The philosophy is that once a user has a hold of the configuration pointer it is okay for him to use it even if the configuration changes. From the user's perspective all he wants is a non-changing snapshot of the configuration. Inktomi recommends that you use `INKConfigSet` for all global data updates.

Here's how the interface works:

```
/* Assume that you have previously defined a plugin configuration
 * data structure named ConfigData, along with its constructor
 * plugin_config_allocator () and its destructor
 * plugin_config_destructor (ConfigData *data)
 */
ConfigData *plugin_config;

/* You will need to assign plugin_config a unique identifier of type
 * unsigned int. It is important to initialize this identifier to zero
 * (see the documentation of the  function).
```

```
 */
static unsigned int   my_id = 0;

/* You will need an INKConfig pointer to access a snapshot of the
 * current plugin_config.
 */
INKConfig config_ptr;

/* Initialize plugin_config. */
plugin_config = plugin_config_allocator();

/* Assign plugin_config an identifier using INKConfigSet. */
my_id = INKConfigSet (my_id, plugin_config, plugin_config_destructor);

/* Get a snapshot of the current configuration using INKConfigGet. */
config_ptr = INKConfigGet (my_id);

/* With an INKConfig pointer to the current configuration, you can
 * retrieve the configuration's current data using INKConfigDataGet.
 */
plugin_config = (ConfigData*) INKConfigDataGet (config_ptr);

/* Do something with plugin_config here. */

/* When you are done with retrieving or modifying the plugin data, you
 * release the pointers to the data with a call to INKConfigRelease.
 */
INKConfigRelease (my_id, config_ptr);

/* Any time you want to modify plugin_config, you must repeat these
 * steps, starting with
 * my_id = INKConfigSet (my_id,plugin_config, plugin_config_destructor);
 * and continuing up to INKConfigRelease.
 */
```

The configuration functions are:

- INKConfigDataGet
- INKConfigGet
- INKConfigRelease
- INKConfigSet

CHAPTER 12

Actions Guide

This chapter contains:

Actions

An action is a handle to an operation initiated by a plugin which has not yet completed. For example, when a plugin connects to a remote server it uses the call `INKNetConnect` which takes an `INKCont` as an argument to call back when the connection is established. `INKNetConnect` might not call the continuation back immediately and will return an `INKAction` structure which the caller can use to cancel the operation. Cancelling the operation does not necessarily mean that the operation will not occur, but that the continuation passed in to the operation will not be called back. In the above example, the connection might still occur if the action is cancelled, but the continuation that initiated the connection would not be called back when that occurred.

It is possible that the connection, in the preceding example, will complete and callback the continuation before `INKNetConnect` returns. If this occurs `INKNetConnect` will return a special action which will cause `INKActionDone` to return `1`. Basically this is specifying that the operation has already completed. There is no point in trying to cancel the operation. Note that an action will never change from non-completed to completed. When the operation actually succeeds and the continuation is called back it is up to the continuation to zero out its action pointer to indicate to itself that the operation succeeded.

The asynchronous nature of all operations in Traffic Edge necessitates actions. You should notice from the above discussion that once a call to a function like `INKNetConnect` is made by a continuation and that function returns a valid action (`INKActionDone` returns `0`) then it is not safe for the continuation to do anything else except return from its handler function. It is not safe to modify or examine the continuation's data since the continuation may have already been destroyed.

Here is an example of a typical usage of an action:

```
#include “InkAPI.h”
static int
handler (INKCont contp, INKEvent event, void *edata)
{
   if (event == INK_EVENT_IMMEDIATE) {
      INKAction actionp = INKNetConnect (contp, 127.0.0.1, 9999);
      if (!INKActionDone (actionp)) {
```

```
    INKContDataSet (contp, actionp);
    } else {
           /* we've already been called back... */
    return 0;
    }
    } else if (event == INK_EVENT_NET_CONNECT) {
       /* net connection succeeded */
       INKContDataSet (contp, NULL);
       return 0;
    } else if (event == INK_EVENT_NET_CONNECT_FAILED) {
    /* net connection failed */
    INKContDataSet (contp, NULL);
    return 0;
    }
   return 0;
}

void
INKPluginInit (int argc, const char *argv[])
{
    INKCont contp;

    contp = INKContCreate (handler, INKMutexCreate ());

    /* We don't want to call things out of INKPluginInit
       directly since it is called before the rest of the
       system is initialized. We'll simply schedule an event
       on the continuation to occur as soon as the rest of
       the system is started up. */
    INKContSchedule (contp, 0);
}
```

The preceding example shows a simple plugin which creates a continuation and schedules it to be called immediately. When the plugin's handler function is called the first time the event will be `INK_EVENT_IMMEDIATE`. The plugin then tries to open a net connection to port 9999 on localhost (127.0.0.1). I've left the IP description in dot notation to make it clearer what is going on. Please note that the above won't actually compile until the IP address is modified. The action returned from `INKNetConnect` is examined by the plugin. If the operation has not completed the plugin stores the action in its continuation. Otherwise the plugin knows it has already been called back and there is no reason to store the action pointer.

A final question might be why would a plugin want to cancel an action. In the above example a valid reason would be to place a time limit on how long it takes to open a connection. The plugin could schedule itself to get called back in 30 seconds and then

initiate the net connection. If the time-out expires first then the plugin would cancel the action. The following sample code implements this:

```
#include “InkAPI.h”
static int
handler (INKCont contp, INKEvent event, void *edata)
{
   switch (event) {
   case (INK_EVENT_IMMEDIATE):
      INKContSchedule (contp, 30000);
      INKAction actionp = INKNetConnect(contp, 127.0.0.1, 9999);
      if (!INKActionDone (actionp)) {
         INKContDataSet (contp, actionp);
      } else {
         /* we’ve already been called back ... */
      }
      break;

   case (INK_EVENT_TIMEOUT):
      INKAction actionp = INKContDataGet (contp);
      if (!INKActionDone(actionp)) {
         INKActionCancel (actionp);
      }
      break;

   case (INK_EVENT_NET_CONNECT):
   /* net connection succeeded */
      INKContDataSet (contp, NULL);
      break;

   case (INK_EVENT_NET_CONNECT_FAILED):
   /* net connection failed */
      INKContDataSet (contp, NULL);
      break;

   }
   return 0;
}

void
INKPluginInit (int argc, const char *argv[])
{
    INKCont contp;
```

```
    contp = INKContCreate (handler, INKMutexCreate ());

    /* We don't want to call things out of INKPluginInit
       directly since it is called before the rest of the
       system is initialized. We'll simply schedule an event
       on the continuation to occur as soon as the rest of
       the system is started up. */
    INKContSchedule (contp, 0);
}
```

The action functions are:

✔*"INKActionCancel" on page 209*

✔*"INKActionDone" on page 210*

Hosts Lookup API

The hosts lookup allows plugins to ask Traffic Edge to do a host lookup of a host name. This is in some way similar to a DNS lookup.

The hosts lookup functions are:

✔*"INKHostLookup" on page 210*

✔*"INKHostLookupResultIPGet" on page 211*

CHAPTER 13

IO Guide

This chapter contains:

Vconnections

The vconnection functions allow you to schedule and obtain and modify information about vconnections.

The vconnection user's view

To use a vconnection, a user first needs to get a handle to one. This is usually accomplished by having it handed to the user or the user issuing a call which creates a vconnection such as `INKNetConnect`. In the case of transform plugins, plugin creates a transformation vconnection using `INKTransformCreate`, and accesses the output vconnection using `INKTransformOutputVConnGet`.

Once the user has a handle to a vconnection he can then issue a read or write call. It's important to note that not all vconnections support both reading and writing. As of yet, there has not been a need to query a vconnection ask to whether it can perform a read or write operation. That ability is obvious from context.

To issue a read or write operation a user calls `INKVConnRead` or `INKVConnWrite`. These two operations both return VIO (`INKVIO`). The VIO describes the operation being performed and how much progress has been made.

Transform plugins initiate output to the downstream vconnection by calling `INKVConnWrite`.

A vconnection read or write operation is different from a normal Unix read(2) or write(2) operation in that the operation can specify more data to be read or written than exists in the buffer handed to the operation. For example, it is typical to issue a read for `INT_MAX` (4 billion) bytes from a network vconnection in order to read all the data from the network connection until we reach the end of stream. Contrast this to the usual Unix fashion of issuing repeated calls to read(2) until one of them finally returns `0` indicating the end of stream was reached. (Yes, the underlying implementation of vconnections on Unix still issues those calls to read(2), but the interface does not expose that detail).

A given vconnection can have at most one read operation and one write operation being performed on it. This is restricted both by design and common sense. If two write operations were to be performed on a single vconnection the user would not be able to specify which one should occur first and the output would occur in an intermingled fashion. Note that both a read operation and a write operation can happen on a single vconnection at the same time. The restriction is on more than one operation of a given type.

One issue that should be obvious is that the buffer passed to `INKVConnRead` and `INKVConnWrite` won't be large enough. There is no reasonable way to make a buffer that can hold `INT_MAX` (4 billion) bytes. The secret is that vconnections engage in a protocol whereby they signal their user (the continuation passed to `INKVConnRead` and `INKVConnWrite`) that they have emptied out the buffers passed to them and are ready for more data. When this occurs it is up to the user to add more data to the buffers (or wait for more data to be added) and then wake up the vconnection by calling *`INKVIOReenable`* on the VIO describing the operation. *`INKVIOReenable`* specifies that the buffer for the operation has been modified and that the vconnection should reexamine it to see if it can make further progress.

The null transform plugin gives an example of how this is done. First, here is the prototype of `INKVConnWrite`:

```
INKVIO INKVConnWrite (INKVConn connp, INKCont contp, INKIOBufferReader
   readerp, int nbytes)
```

Where the *`connp`* is the vconnection that the user is writing to, and *`contp`* is the "user" – it is the continuation that *`connp`* calls back when it has emptied out its buffer and is ready for more data.

The call made in the null transform plugin is:

```
INKVConnWrite (output_conn, contp, data->output_reader, INKVIONBytesGet
   (input_vio));
```

In this example, `contp` is the transformation vconnection, which is writing to the output vconnection. The number of bytes to be written is obtained from the `input_vio` by `INKVIONBytesGet`.

When a vconnection calls back its user to indicate that it wants more data or when some other condition has occurred, it issues a call to `INKContCall` and passes one of the following values as the event parameter and the `INKVIO` describing the operation as the data parameter.

Event parameter value	Description
INK_EVENT_ERROR	Indicates that an error has occurred on the vconnection. This will happen for network IO if the underlying read(2) or write(2) call return an error.
INK_EVENT_VCONN_READ_READY	The vconnection has placed data in the buffer passed to an `INKVConnRead` operation and it would like to do more IO but the buffer is now full. When the user consumes the data from the buffer it should re-enable the VIO to indicate to the vconnection that the buffer has been modified.

Event parameter value	Description
INK_EVENT_VCONN_WRITE_READY	The vconnection has removed data from the buffer passed to an `INKVConnWrite` operation and it would like to do more IO but the buffer does not have enough data in it. When the user places more data in the buffer he should re-enable the VIO to indicate to the vconnection that the buffer has been modified.
INK_EVENT_VCONN_READ_COMPLETE	The vconnection has read all the bytes specified by an `INKVConnRead` operation. The vconnection can now be used to initiate a new IO operation.
INK_EVENT_VCONN_WRITE_COMPLETE	The vconnection has written all the bytes specified by an `INKVConnWrite` operation. The vconnection can now be used to initiate a new IO operation.
INK_EVENT_VCONN_EOS	An attempt was made to read past the end of the stream of bytes during the handling of an `INKVConnRead` operation. This event occurs when the number of bytes available for reading from a vconnection is less than the number of bytes the user specifies should be read from the vconnection in a call to `INKVConnRead`. A common case where this occurs is when the user specifies that `INT_MAX` bytes are to be read from network connection.

The null transform plugin's transformation, for example, receives `INK_EVENT_VCONN_WRITE_READY` and `INK_EVENT_VCONN_WRITE_COMPLETE` events from the downstream vconnection as a result of the call to `INKVConnWrite`.

When the user is finished using a vconnection he needs to call `INKVConnClose` or `INKVConnAbort`. Both calls indicate that the vconnection can destroy itself but `INKVConnAbort` should be used when the connection is being closed abnormally. After a call to `INKVConnClose` or `INKVConnAbort` the user will not be called back by the vconnection again.

Sometimes it's desirable to simply close down the write portion of a connection while keeping the read portion open. This can be accomplished using the `INKVConnShutdown` function which will shutdown either the read or write portion of a vconnection. Shutdown means that the vconnection will no longer call back the user with events for the portion of the connection shutdown. For example, if the user shuts down the write portion of a connection he will no longer get `INK_EVENT_VCONN_WRITE_READY` or `INK_EVENT_VCONN_WRITE_COMPLETE` events.

In the null transform plugin, the write operation is shut down with a call to `INKVConnShutdown`.

For a description of how vconnections are used in transformation plugins, see *Writing content transform plugins, on page 41*.

The vconnection functions are:

- INKVConnAbort
- INKVConnClose
- INKVConnClosedGet (used for Transformations only)
- INKVConnCreate
- INKVConnRead
- INKVConnReadVIOGet
- INKVConnShutdown
- INKVConnWrite
- INKVConnWriteVIOGet

Net VConnections

A network vconnection (netvconnection) is a wrapper around a TCP socket that allows the socket to work within the Traffic Edge vconnection framework. See *Vconnections, on page 121* for more information about the Traffic Edge abstraction for doing asynchronous IO.

The net vconnection functions are:

- *INKNetAccept, on page 214*
- *INKNetConnect, on page 214*

Transformations

The vconnection implementor's view

A VConnection implementor writes only transformations. All other VConnections (net VConnections and cache VConnections) are implemented in iocore. As mentioned earlier, a given vconnection can have at most one read operation and one write operation being performed on it. The vconnection user gets information about the operation being performed by examining the VIO returned by a call to `INKVConnRead` or `INKVConnWrite`. The implementor, in turn, gets a handle on the VIO operation by examining the VIO returned by `INKVConnReadVIOGet` or `INKVConnWriteVIOGet`. (Recall that every vconnection created through the Traffic Edge API has an associated read VIO and write VIO even if it only supports reading or writing.)

For example, the null transform plugin's transformation examines the input VIO by calling

```
input_vio = INKVConnWriteVIOGet (contp);
```

Where `contp` is the transformation.

A vconnection is a continuation, which means it has a handler function that gets run when an event is sent to it, or more accurately, when an event that was sent to it is received. It is the handler function's job to examine the event, the current state of its read VIO and write

VIO and any other internal state the vconnection might have and potentially make some progress on the IO operations.

It is common for the handler function for all vconnections to look similar. Their basic form looks something like the following code fragment.

```
int
vconnection_handler (INKCont contp, INKEvent event, void *edata)
{
    if (INKVConnClosedGet (contp)) {
        /* Destroy any vconnection specific data here. */
        INKContDestroy (contp);
        return 0;
   } else {
        /* Handle the incoming event */
   }
}
```

This code fragment basically shows that many vconnections simply want to destroy themselves when they are closed. However, the situation might also require the vconnection to do some cleanup processing which is why `INKVConnClose` does not simply just destroy the vconnection.

Vconnections are state machines which are animated by the events they receive. An event is sent to the vconnection whenever an `INKVConnRead`, `INKVConnWrite`, `INKVConnClose`, `INKVConnShutdown` or `INKVIOReenable` call is performed. `INKVIOReenable` indirectly references the vconnection through a back-pointer in the VIO structure to the vconnection. The vconnection itself only knows what call was performed by examining its state and the state of its VIOs. For example, when `INKVConnClose` is called, the vconnection will be sent an immediate event (`INK_EVENT_IMMEDIATE`). For every event the vconnection receives, it needs to check its closed flag to see if it has been closed. Similarly, when `INKVIOReenable` is called, the vconnection is sent an immediate event. So for every event the vconnection receives, it needs to check its VIOs to see if the buffers have been modified to a state where it can continue processing one of its operations.

Lastly, a vconnection is likely the user of other vconnections. It also receives events as the user of these other vconnections. When it receives such an event, like `INK_EVENT_VCONN_WRITE_READY`, it might just enable another vconnection that is writing into the buffer used by the vconnection reading from it. The above description is merely intended to give the overall idea for what a vconnection needs to do.

Transformation VConnection

A transformation is a specific type of vconnection which supports a subset of the vconnection functionality that allows one or more transformations to be chained together. See *Transformations, on page 42* for a description of how to use transformations in transformation plugins.

A transformation is a specific type of vconnection which supports a subset of the vconnection functionality that allows one or more transformations to be chained together. A transformation sits as a bottleneck between an input data source and an output data sink which enables it to view and modify all the data passing through it. Some transformations simply scan the data and pass it on. A common transformation is to compress the data in some manner.

A transformation can modify either the data stream being sent to an HTTP client (e.g. the document) or the data stream being sent from an HTTP client (e.g. post data). To do so the transformation should hook on to one of these hooks:

✔`INK_HTTP_REQUEST_TRANSFORM_HOOK`

✔`INK_HTTP_RESPONSE_TRANSFORM_HOOK`

Note that because a transformation is intimately associated with a given transaction that it is only possible to add the hook to the transaction hooks and not to the global or session hooks. Transformations reside in a chain and their ordering is very simply determined. Transformations adding themselves to the chain are appended to it.

Data is passed in to the transformation by initiating a vconnection write operation on the transformation. The consequence of this design is that a transformation *must* support the vconnection write operation. In other words, your transformation must expect an upstream vconnection to write data to it. The transformation has to read the data, consume it, and tell the upstream vconnection that it is finished by send it an `INK_EVENT_WRITE_COMPLETE` event.

transformations must consume all upstream data before closing

Transformations cannot send `INK_EVENT_VCONN_WRITE_COMPLETE` to the upstream vconnection unless they are finished consuming all incoming data. If `INK_EVENT_VCONN_WRITE_COMPLETE` is sent prematurely, certain internal Traffic Edge data structures will not be deallocated, causing a memory leak.

How to make sure that all incoming data is consumed:

✔Make sure that after reading or copying data, you consume the data and increase the value of ndone for the input VIO, as in the following example taken from `null-transform.c`:

```
INKIOBufferCopy (INKVIOBufferGet (data->output_vio),
INKVIOReaderGet (input_vio), towrite, 0);
/* Tell the read buffer that we have read the data and are no
longer interested in it. */
INKIOBufferReaderConsume (INKVIOReaderGet (input_vio),
towrite);
/* Modify the input VIO to reflect how much has been read.*/
INKVIONDoneSet (input_vio, INKVIONDoneGet (input_vio) +
towrite);
```

✔Before sending `INK_EVENT_VCONN_WRITE_COMPLETE`, your transformation should check the numbe of bytes remaining in the upstream vconnection's write VIO (input VIO) using the function `INKVIONTodoGet (input_vio)`. This value should go to zero when all of the upstream data is consumed (`INKVIONTodoGet = nbytes - ndone`). Do not send `INK_EVENT_VCONN_WRITE_COMPLETE` events if `INKVIONTodoGet` is greater than zero.

The transformation passes data out of itself by using the output vconnection retrieved by `INKTransformOutputVConnGet`. Immediately before Traffic Edge initiates the write operation which inputs data into the transformation it sets the output vconnection to the next transformation in the chain of transformations or to a special terminating transformation if this is the last transformation in the chain. Since the transformation is handed ownership of the output vconnection it *must* close it at some point in order for it to be de-allocated.

All of the transformations in a transformation chain share the transaction's mutex. This small restriction (enforced by `INKTransformCreate`) removes many of the locking complications of implementing general vconnections. For example, a transformation does not have to grab its write VIO mutex before accessing its write VIO since it knows it already holds the mutex.

The transformation functions are:

- *INKTransformCreate, on page 220*
- *INKTransformOutputVConnGet, on page 221*

VIOs

A VIO or virtual IO is a description of an in progress IO operation. The VIO data structure is used by vconnection users to determine how much progress has been made on a particular IO operation and to re-enable an IO operation when it stalls due to buffer space. VIOs are used by vconnection implementors to determine the buffer for an IO operation, to determine how much work to do on the IO operation and to determine which continuation to call back when progress on the IO operation is made.

The `INKVIO` data structure itself is opaque, but it might have been defined as follows:

```
typedef struct {
    INKCont continuation;
    INKVConn vconnection;
    INKIOBufferReader reader;
    INKMutex mutex;
    int nbytes;
    int ndone;
} *INKVIO;
```

The functions below simply access and modify various parts of the data structure.

- INKVIOBufferGet
- INKVIOVConnGet
- INKVIOContGet
- INKVIOMutexGet
- INKVIONBytesGet
- INKVIONBytesSet
- INKVIONDoneGet
- INKVIONDoneSet
- INKVIONTodoGet
- INKVIOReaderGet
- INKVIOReenable

IO buffers

The IO buffer data structure is the building block of the *vconnection* abstraction. An IO buffer (`INKIOBuffer`) is composed of a list of buffer blocks which in turn point to buffer data. Both the buffer block (`INKIOBufferBlock`) and buffer data (`INKIOBufferData`) data structures are reference counted so that they can reside in multiple buffers at the same time. This makes it extremely efficient to copy data from one IO buffer to another using `INKIOBufferCopy` since Traffic Edge only needs to copy pointers and adjust reference counts appropriately and not actually copy any data.

The IO buffer abstraction provides for a single writer and multiple readers. In order for the readers to have no knowledge of each other, they manipulate IO buffers through the `INKIOBufferReader` data structure. Since only a single writer is allowed, there is no corresponding `INKIOBufferWriter` data structure. The writer simply modifies the IO buffer directly.

The IO buffer functions are:

Refer to the sample code in the description of *INKIOBufferBlockReadStart, on page 225* for a sample that illustrates how to use IOBuffers.

- The INKIOBufferReader data structure keeps track of how much data in the INKIOBuffer has been read. It has an offset number of bytes which is the current start point of a particular buffer reader. (For every read operation on an INKIOBuffer, you must allocate an INKIOBufferReader).
- Note that the bytes that already have been read may or may not be freed within the INKIOBuffer. You have to call INKIOBufferConsume to consume bytes that have been read. See the sample code on page 215. See also the output-hdr.c sample plugin that Chris Cooper sent.

Guide to the cache API

The cache API lets plugins read, write, and remove objects in the Traffic Edge cache. All cache APIs are keyed by an object called an `INKCacheKey`. Cache keys are created via `INKCacheKeyCreate`. Keys are destroyed via `INKCacheKeyDestroy`. Use `INKCacheKeyDigestSet` to set the hash of the cache key.

Note that the cache APIs differentiates between HTTP data and plugin data. The cache APIs do not allow you to write HTTP docs in the cache. You can only write plugin specific data which is a specific type of data which is different from the HTTP type.

Example:

```
  const unsigned char *key_name = "example key name";


    INKCacheKey key;
    INKCacheKeyCreate (&key);
    INKCacheKeyDigestSet (key, (unsigned char *) key_name ,
strlen(key_name));
    INKCacheKeyDestroy (key);
```

How to do a cache read

INKCacheRead does not really read, it is used for lookups. See the sample Protocol plugin. The possible callback events include:

- `INK_EVENT_CACHE_OPEN_READ` - indicating that the lookup was successful, the data passed back along with this event is a cache vconnection that may be used to initiate a read on this keyed data.
- `INK_EVENT_CACHE_OPEN_READ_FAILED` - indicating that the lookup was unsuccessful. Reasons for this event include: another continuation could be writing to that cache location, or the cache key may not refer to a cached resource. Data payload for this event indicates the possible reason for the read failing (`INKCacheError`).

How to do a cache write

Use `INKCacheWrite` to write to a cache. See the sample Protocol plugin. The possible callback events include:

- `INK_EVENT_CACHE_WRITE_READ` - indicating that the lookup was successful, the data passed back along with this event is a cache vconnection that may be used to initiate a write to the cache.
- `INK_EVENT_CACHE_OPEN_WRITE_FAILED` - The event is returned if another continuation is currently writing to this location in the cache. Data payload for this event indicates the possible reason for the write failing (`INKCacheError`).

How to do a cache remove

Use `INKCacheRemove` to remove items from the cache. The possible callback events include:

- `INK_EVENT_CACHE_REMOVE` - item was removed. There is not data payload for this event.
- `INK_EVENT_CACHE_REMOVE_FAILED` - indicating that the cache was unabled to remove the item idetified by the cache key. Data indicates the reason why the removed failed (`INKCacheError`).

Errors

Errors as to why various cache operations failed are indicated by INKCacheError (enumeration) as follows:

- `INK_CACHE_ERROR_NO_DOC` - key does not match a cached resource.
- `INK_CACHE_ERROR_DOC_BUSY` - e.g, another continuation could be writing to that cache location.
- `INK_CACHE_ERROR_NOT_READY` - the cache is not ready.

Example

In the example below, suppose we have a cache hit and the cache returns a vconnection for us to read the document from the cache. To do this, we have to prepare a buffer

(*cache_bufp*) to hold the document. Meanwhile, we would use INKVConnCachedObjectSizeGet to tell us the actual size of the document (*content_length*). After, we would issue INKVConnRead to read the document with the total data length required being *content_length*. Assume the following data:

```
INKIOBuffer       cache_bufp = INKIOBufferCreate ();
INKIOBufferReader cache_readerp = INKIOBufferReaderAlloc (out_bufp);
INKVConn          cache_vconnp = NULL;
INKVIO            cache_vio = NULL;
int               content_length = 0;
```

In the INK_CACHE_OPEN_READ handler;

```
cache_vconnp = (INKVConn) data;
INKVConnCachedObjectSizeGet (cache_vconnp, &content_length);
cache_vio = INKVConnRead (cache_vconn, contp, cache_bufp,
content_length);
```

In the INK_EVENT_VCONN_READ_READY handler:

```
(usual VCONN_READ_READY handler logic)
int nbytes = INKVIONBytesGet (cache_vio);
int ntodo  = INKVIONTodoGet (cache_vio);
int ndone  = INKVIONDoneGet (cache_vio);
(consume data in cache_bufp)
INKVIOReenable (cache_vio);
```

Do not try to get continuations or vios from INKVConn objects for Cache VConnections. Also note that the following APIs can only be used on Transformation VConnections and must not be used on Cache or Net VConnections:

- INKVConnWriteVIOGet
- INKVConnReadVIOGet
- INKVConnClosedGet

APIs such as INKVConnRead, INKVConnWrite, INKVConnClose, INKVConnAbort and INKVConnShutdown can be used on any kind of VConnections.

When you are done:

```
INKCacheKeyDestroy (key);
```

CHAPTER 14

Plugin Management

This chapter decribes:

- *"Setting up a plugin management interface" on page 131.*

  You can add your own HTML information pages or CGI forms to the Traffic Manager UI. Traffic Manager can send configuration information from a CGI form to your plugin.

- *"Reading Traffic Edge settings and statistics" on page 132.*

  Using the functions in this chapter, plugins can read Traffic Edge configuration settings and statistics from the `records.config` file.

- *"Accessing installed plugin files" on page 132.*

  Have plugins access related files in the plugin installation directory, and make sure that your plugins are preserved during Traffic Edge upgrades.

- *"Licensing your plugin" on page 133.*

- *"Guide to the logging API" on page 135.*

  The logging API enables your plugin to log text entries in a custom Traffic Edge log file. This section gives a basic overview of the logging interface.

Setting up a plugin management interface

To set up a plugin management interface, follow these steps:

1. Create your interface. It must be browser-based, since it is accessed through the Traffic Manager UI. Your interface can be static or dynamic. If you are using a dynamic interface, your CGI form submission must set the `INK_PLUGIN_NAME` variable to be the name of your plugin, as it is entered in the `INKMgmtUpdateRegister` function.
2. Note the location of your interface files.
3. Use the `INKMgmtUpdateRegister` function in your plugin. It should be part of `INKPluginInit`.

The `INKMgmtUpdateRegister` function does two things:

- Informs Traffic Manager of the location of your interface, so that the links to your interface appear in the Traffic Manager UI
- If you have a dynamic interface, it sets up a callback to your plugin whenever configuration information is submitted through the interface

Reading Traffic Edge settings and statistics

Your plugin might need to know information about Traffic Edge's current configuration and performance. The functions described in this section read this information from the Traffic Edge `records.config` file. Configuration settings are stored in `CONFIG` variables and statistics are stored in `PROCESS` variables.

Caution

Not all `CONFIG` and `PROCESS` variables in `records.config` are relevant to Traffic Edge's configuration and statistics. Retrieve only the `records.config` variables that are documented in the *Traffic Edge Administrator's Guide*.

Four result types

To retrieve a variable, you need to know its type (`int`, `counter`, `float`, or `string`). Plugins store the `records.config` values as an `INKMgmtInt`, `INKMgmtCounter`, `INKMgmtFloat`, or `INKMgmtString`. You can look up `records.config` variable types in the *Traffic Edge Administrator's Guide*.

Depending on the result type, use `INKMgmtIntGet`, `INKMgmtCounterGet`, `INKMgmtFloatGet`, or `INKMgmtStringGet` to obtain the variable value.

See the example for *"INKMgmtIntGet" on page 234.*.

The `INKMgmt*Get` functions are:

- *"INKMgmtCounterGet" on page 233.*
- *"INKMgmtFloatGet" on page 234.*
- *"INKMgmtIntGet" on page 234.*
- *"INKMgmtStringGet" on page 234.*

Accessing installed plugin files

Your plugin might rely on files in addition to its source code, such as configuration files. When you upgrade Traffic Edge, you might need to make sure your plugin is always able to find its associated files. The mechanism for preserving relative file locations with upgrades is the following:

- Make sure all plugins are contained in their own directories within the plugin directory.
- The plugin directory path is specified in the Traffic Edge `records.config` file variable `proxy.config.plugin.plugin_dir`. This path is relative to the Traffic Edge install directory. The default value is `config/plugin`.
- Make sure all plugins are listed in the `plugin.db` file. For each plugin, this file contains the plugin name, object file, license key, file name(s), and directory relative to the plugin directory.
- When Traffic Edge is upgraded, the Traffic Edge installation program looks at the `plugin.db` file to see which plugins to copy over to the new Traffic Edge installation, and what the appropriate object files, license keys, additional files, and directories should be.

The format of the `plugin.db` file is as follows:

```
[name of your plugin]
```

```
Object=[name of plugin's shared object file
License=[license key]
Dir=[name of any directories to be copied over]
```

notes about plugin.db format

- Entries in `plugin.db` are case-sensitive.
- Do not include white spaces in your entries. For example, the following line is incorrect:

  `Object = plugin.so`

  The correct entry would be:

  `Object=plugin.so`

For example, suppose that you have a blacklist plugin in the plugin directory. Its object file is `Blacklist.so` and it has some user interface files (images and HTML files) in the `Blacklist/ui` directory. To make sure that the blacklist plugin is upgraded properly, `plugin.db` needs the following entry:

```
[Blacklist plugin]
Object=Blacklist.so
License=ABCD0123456789
Dir=Blacklist/ui
```

In this example, if all of the necessary files and directories are in the Blacklist directory, you could simply specify `Dir = Blacklist`.

This means that the Blacklist image and HTML files are always located in:

`<Traffic Edge install directory>/<plugin directory>/Blacklist/ui`

Your plugin might need to specify the absolute location of its associated files. The following functions provide the Traffic Edge install directory path and plugin directory path:

- *"INKInstallDirGet" on page 235.*
- *"INKPluginDirGet" on page 235.*

Licensing your plugin

When installing a plugin which requires a license, the plugin.db must be updated. This file contains the license keys for the plugins. At load time, Traffic Edge reads the key in the plugin.db file and checks their validity. If a key is not valid, the plugin is not executed.

Format of plugin.db

```
# comments start by a '#' character
[plugin_name]
Object=plugin.so
License=Key
```

Be careful with the syntax:

- Object is with an uppercase 'O'

  ex.: `object=plugin.so` is bad

- License is with an uppercase 'L'

  ex. `license=key` is bad

- No blank between '=' and value.

  ex.: `Object = plugin.so` is bad

- No blank after the value.

  ex. `License=Key` is bad

Setting up licensing

Set up licensing in these steps:

1. Develop your plugin, using the `INKPluginLicenseRequired` function.
2. Create an installation program for your plugin. The installation program must update both `plugin.config` and `plugin.db`. When your plugin customer installs the plugin, the program should ask the customer for the license key.
3. Use the `gen_key` tool to generate license keys. You can generate different keys for different customers, and you can set expiration dates for each key.
4. Distribute your plugin together with license key to customers.

When the customer installs the plugin, the installation program should ask for the license key. The installation program should then make an entry in `plugin.db` and `plugin.config` for the new plugin. When the customer runs the plugin, Traffic Edge checks the license key. If it passes, Traffic Edge then calls `INKPluginInit`.

Example

- You have a plugin *`filtering`*, implemented in object *`filtering.so`*
- You generate a key for your plugin *`filtering`* by using:

  `gen_key` *`filtering ABCDE 03312002`*

- The key generated by `gen_key` is:

  `ABCDE2E01E07D95`

- You must update `plugin.db` and add the following lines:

```
[filtering]
Object=filtering.so
License=ABCDE2E01E07D95
```

The following function is used to license your plugin:

- *"INKPluginLicenseRequired" on page 235.*

Generating a license key

The `gen_key` tool generates a license key based on your plugin name (which must match the plugin name entered in the `plugin.db` file), an expiration date, and a customer ID (so

that you can give different license keys to different customers). You can specify an expiration date of `0` which means that the plugin never expires.

▼ Running the gen_key tool

1 On Unix, `cd` to the `sdk/tools` directory in your SDK package. On NT, open a DOS command window and `cd` to the `sdk/tools` directory.

2 Enter the following:

```
gen_key plugin_name ID expiration
```

- * plugin_name is the name of the plugin and it needs to match the name specified in plugin.db
- ID is a string of 5 alphanumeric characters, used to identify different customers
- expiration is the expiration date of the plugin in the following format:

  `mmddyyyy`

  For example, `03312001` for March 31, 2001. Use `0` for no expiration.

Guide to the logging API

The logging API lets your plugin log entries in a custom text log file. You create the file with the call `INKTextLogObjectCreate`. The log file is part of Traffic Edge's logging system. By default, the log file is stored in the logging directory. Once you have created the log object, you can set log properties.

The logging API enables you to:

✔Establish a custom text log for your plugin. See *"INKTextLogObjectCreate" on page 240.*.

- Set the log header for your custom text log. See *"INKTextLogObjectHeaderSet" on page 241.*
- Enable or disable rolling your custom text log. See *"INKTextLogObjectRollingEnabledSet" on page 242.*
- Set the rolling interval in seconds for your custom text log. See *"INKTextLogObjectRollingIntervalSecSet" on page 242.*
- Set the rolling offset for your custom text log. See *"INKTextLogObjectRollingOffsetHrSet" on page 243.*

✔Write text entries to the custom text log. See *"INKTextLogObjectWrite" on page 243.*.

✔Flush the contents of the custom text log's write buffer to disk. See *"INKTextLogObjectFlush" on page 243.*.

✔Destroy custom text logs when you are done with them. See *"INKTextLogObjectDestroy" on page 244.*.

Here is how the logging API is used in the `blacklist-1.c` sample plugin. See *"Sample Source Code" on page 245.* for complete source code.

1 A new log file is defined as a global variable.

```
static INKTextLogObject log;
```

2 In `INKPluginInit`, a new log object is allocated:

```
log = INKTextLogObjectCreate("blacklist",
INK_LOG_MODE_ADD_TIMESTAMP,
    NULL, &error);
```

The new log is named `blacklist.log`. Each entry written to the log will have a timestamp. The `NULL` argument specifies that the new log does not have a log header. The `error` argument stores the result of the log creation. If the log is created successfully, `error` is equal to `INK_LOG_ERROR_NO_ERROR`.

3 After creating the log, the plugin makes sure that the log was created successfully:

```
if (!log) {
    printf("Blacklist plugin: error %d while creating log\n",
error);
}
```

4 The `blacklist-1` plugin matches the host portion of the URL in each client request with a list of blacklisted sites stored in the array `sites[]`:

```
for (i = 0; i < nsites; i++) {
    if (strncmp (host, sites[i], host_length) == 0) {
```

If the host matches one of the blacklisted sites, say `sites[i]`, then the plugin writes a blacklist entry to `blacklist.log`:

```
     if (log) {
       INKTextLogObjectWrite(log, "blacklisting site: %s",
sites[i]);
```

The format of the log entry is :

`<timestamp> blacklisting site:` *`sites[i]`*

The log is not flushed or destroyed in the `blacklist-1` plugin. It lives for the life of the plugin.

CHAPTER 15

Adding Statistics

This chapter describes how to add statistics to your plugins. Statistics can be coupled or uncoupled. Coupled statistics are quantities that are related and must be updated together. The Traffic Edge API statistics functions add your plugin's statistics to the Traffic Edge statistics system. You can view your plugin statistics as you would any Traffic Edge statistic, using Traffic Line (Traffic Edge's command line interface). This chapter contains the following topics:

Uncoupled statistics

A statistic is an object of type `INKStat`. The value of the statistic is of type `INKStatType`. The possible `INKStatTypes` are:

- `INKSTAT_TYPE_INT64`
- `INKSTAT_TYPE_FLOAT`

There is *no* `INKSTAT_TYPE_INT32`.

To add uncoupled statistics, follow these steps:

1. Declare your statistic as a global variable in your plugin. For example:

   ```
   static INKStat my_statistic;
   ```

2. In `INKPluginInit`, create new statistics using `INKStatCreate`.

   When you create a new statistic, you need to give it an "external" name that the Traffic Edge command line interface (Traffic Line) uses to access the statistic. For example:

   ```
   my_statistic = INKStatCreate ("my.statistic",
   INKSTAT_TYPE_INT64);
   ```

3. Modify (increment, decrement, or other modification) your statistic in plugin functions.

Coupled statistics

Use coupled statistics for quantities that are related and must be updated jointly. As a very simple example, suppose that you have three statistics: `sum`, `part_1` and `part_2`, and they must always preserve the relationship that `sum = part_1 + part_2`. If you update

part_1 without updating sum at the same time, the equation would be untrue. The mechanism for updating coupled statistics jointly is to create local copies of global coupled statistics in the routines that modifiy them. When each local copy is updated appropriately, you do a global update using INKStatsCoupledUpdate. To specify which statistics are related to one another, you establish a coupled statistic category, and make sure that each coupled statistic belongs to the appropriate category. When it is time to do the global update, you specify the category to be updated.

Note The local statistic copy must have a duplicate set of statistics as that of the master copy. Local statistics must also be added to the local statistic category in the same order as their master copy counterparts were added originally.

Here are the steps you needed, followed by an example of code that is taken from the redirect-1.c sample plugin.

▼ **To add coupled statistics:**

1. Declare the global category for your coupled statistics as a global INKCoupledStat variable in your plugin.
2. Declare your coupled statistics as global INKStat variables in your plugin.
3. In INKPluginInit, create a new global coupled category using INKStatCoupledGlobalCategoryCreate.
4. In INKPluginInit, create new global coupled statistics using INKStatCoupledGlobalAdd.

   When you create a new statistic, you need to give it an "external" name that the Traffic Edge command line interface (Traffic Line) uses to access the statistic.
5. In any routine where you want to modify (increment, decrement, or other modification) your coupled statistics, declare local copies of the coupled category and coupled statistics.
6. Then create local copies using INKStatCoupledLocalCopyCreate and INKStatCoupledLocalAdd.
7. Modify the local copies of your statistics. Then to update the global copies jointly, call INKStatsCoupledUpdate.
8. When you are done, you must destroy the all of the local copies in the category using INKStatCoupledLocalCopyDestroy.

Example using the redirect-1.c sample plugin

```
static INKCoupledStat request_outcomes;

static INKStat requests_all;
static INKStat requests_redirects;
static INKStat requests_unchanged;

request_outcomes = INKStatCoupledGlobalCategoryCreate ("request_outcomes");

requests_all = INKStatCoupledGlobalAdd (request_outcomes, "requests.all",
INKSTAT_TYPE_FLOAT);
```

```
requests_redirects = INKStatCoupledGlobalAdd (request_outcomes,
"requests.redirects",
    INKSTAT_TYPE_INT64);
requests_unchanged = INKStatCoupledGlobalAdd (request_outcomes,
"requests.unchanged",
    INKSTAT_TYPE_INT64);


INKCoupledStat local_request_outcomes;
INKStat local_requests_all;
INKStat local_requests_redirects;
INKStat local_requests_unchanged;


local_request_outcomes = INKStatCoupledLocalCopyCreate("local_request_outcomes",
    request_outcomes);
local_requests_all = INKStatCoupledLocalAdd(local_request_outcomes,
"requests.all.local",
    INKSTAT_TYPE_FLOAT);
local_requests_redirects = INKStatCoupledLocalAdd(local_request_outcomes,
    "requests.redirects.local", INKSTAT_TYPE_INT64);
local_requests_unchanged = INKStatCoupledLocalAdd(local_request_outcomes,
    "requests.unchanged.local", INKSTAT_TYPE_INT64);


INKStatFloatAddTo( local_requests_all, 1.0 ) ;
...
INKStatIncrement (local_requests_unchanged);
INKStatsCoupledUpdate(local_request_outcomes);


INKStatCoupledLocalCopyDestroy(local_request_outcomes);
```

Viewing statistics using Traffic Line

To view your plugin's statistics, follow these steps:

1. Make sure you know the name of your statistic (the name used in the `INKStatCoupledGlobalAdd`, `INKStatCreate`, or `INKStatCoupledGlobalCategoryCreate` call).
2. In your `<Traffic Edge>/bin` directory, enter the following:

   `./traffic_line -r` *`the_name`*

CHAPTER 16

Function Reference

This chapter provides a description of each function in the Traffic Edge API. The functions are grouped according to what they do. The following section lists all the function groups. You can look up functions alphabetically in the *Function Index, on page 281*.

List of function groups

Initialization functions

INKPluginInit

Prototype	`void` **`INKPluginInit`** `(int` *`argc`*`, const char *`*`argv[]`*`)`
Arguments	*`argc`* is a count of the number of arguments in the argument vector, *`argv`*. The count is at least one because the first argument in the argument vector is the plugin's name, which must exist in order for the plugin to be loaded. *`argv`* is the vector of arguments. The number of arguments in the vector is *`argc`*, and `argv[0]` always contains the name of the plugin shared library.
Description	This function *must* be defined by all plugins. Traffic Edge calls this initialization routine when it loads the plugin and sets *`argc`* and *`argv`* appropriately based on the values in `plugin.config`.
First release	Traffic Server 3.0

INKPluginRegister

Registers the appropriate SDK version for your plugin.

Prototype	`int` **`INKPluginRegister`** `(INKSDKVersion` *`sdk_version`*`,` `INKPluginRegistrationInfo *`*`plugin_info`*`)`

Arguments	*`sdk_version`* can have the following values: `INK_SDK_VERSION_1_0`, `INK_SDK_VERSION_1_1`, `INK_SDK_VERSION_2_0`, `INK_SDK_VERSION_5_2`.
	`INKPluginRegistrationInfo` is the following struct:
	`typedef struct`
	`{`
	`char *plugin_name;`
	`char *vendor_name;`
	`char *support_email;`
	`} INKPluginRegistrationInfo;`

Description	Registers the appropriate SDK version for your plugin. Use this function to make sure that the version of Traffic Edge on which your plugin is running supports the plugin. See *Modified hello-world that checks Traffic Edge version, on page 20* for usage. **Important**: Previous versions of Traffic Edge are named Traffic Server. Throughout this manual, Traffic Server, Traffic Server 3.0, Traffic Server 3.5, and Traffic Server 5.2 refer to previous versions of Traffic Edge. For version checking, Traffic Edge 1.5 is equivalent to Traffic Server 5.5.
Returns	Returns 0 if the plugin registration fails.
First release	Traffic Server 3.5

INKTrafficServerVersionGet

Returns the version of Traffic Edge running the plugin.

Prototype	`const char* INKTrafficServerVersionGet (void)`
Description	Returns the release version of Traffic Edge running the plugin as a string. See *Modified hello-world that checks Traffic Edge version, on page 20* for usage.
Returns	A pointer to a string of characters that describes the Traffic Edge release version. **Important**: Previous versions of Traffic Edge are named Traffic Server. Throughout this manual, Traffic Server, Traffic Server 3.0, Traffic Server 3.5, and Traffic Server 5.2 refer to previous versions of Traffic Edge. For version checking, Traffic Edge 1.5 is equivalent to Traffic Server 5.5.
First release	Traffic Server 3.5

Debugging functions

The debugging functions are:

INKDebug

Issues debug statements.

Prototype	`void INKDebug (const char *tag, const char *format_str, ...)`
Arguments	*`tag`* is the Traffic Edge parameter that enables Traffic Edge to print out *`format_str`*. `...` is a variable for *`format_str`*.

Description	`INKDebug` prints out the statement `format_str` if debugging is enabled. There are two ways to enable debugging:
	◆ On UNIX systems, run Traffic Edge with the `-Ttag` option. For example, if the tag is `my-plugin`:
	`traffic_server -Tmy-plugin`
	In this case, the debug output goes to `traffic.out`.
	◆ On either UNIX or Windows NT systems, set the following variables in records.config (in the Traffic Edge config directory):
	`proxy.config.diags.debug.enabled INT 1`
	`proxy.config.diags.debug.tags STRING` *`debug-tag-name`*
	In this case, debug output goes to `traffic.out` on UNIX systems, and to `diags.log` on Windows NT systems.

Example	`INKDebug` ("my-plugin", “Starting my-plugin at %d\n”, the_time); The statement “Starting my-plugin at <time>” appears whenever you run Traffic Edge with the my-plugin tag: `traffic_server -Tmy-plugin`
First release	Traffic Server 3.5

INKIsDebugTagSet

Tells you if a particular debug tag is set.

Prototype `int` **`INKIsDebugTagSet`** `(const char *`*`t`*` )`

Description Returns 1 if the debug tag *`t`* is set. You can use this tag to let the Traffic Edge administrator know whether the debug tag is set or not.

Example

```
if ( INKIsDebugTagSet( "demo" ) )
  INKDebug( "init", "The demo tag is set" );
else
  INKDebug( "init", "The demo tag is not set" ) ;
```

In this example if you run Traffic Edge with the `init` tag, it will tell you whether or not the `demo` tag is set. You can run Traffic Edge with more than one debug tag set, by adding the tags to the debug tag variable in records.config, for example:

```
proxy.config.diags.debug STRING init demo
```

Returns 0 if the specified debug tag **is not** set.

1 if the specified debug tag is set.

First release Traffic Server 3.5

INKError

Writes an error to the Traffic Edge error log.

Prototype `void` **`INKError`** `(const char *`*`fmt`*`, ...)`

Arguments *`fmt`* is the `printf` format description.

`...` is the argument for the format description.

Description It is sometimes useful to log messages when errors occur. Traffic Edge has a global error log file to which it writes such messages. The function `INKError` is the API interface to this error log. `INKError` is similar to `printf` except that instead of writing the output to the C standard output, `INKError` writes output to the Traffic Edge error log. One advantage of `INKError` over `printf` is that each call is atomically placed into the error log and is not garbled with other error entries. This is not an issue in single-threaded programs but is a definite nuisance in multi-threaded programs.

Example `INKError` ("couldn't retrieve client request header\n");

First release Traffic Server 3.0

INKAssert

Allows the use of assertion in a plugin.

Prototype `void INKAssert(`*`expression`*`);`

Arguments A boolean expression.

Description If expression is false:

In debug mode, causes the Traffic Edge to print the file name, line number and expression, then to abort.

In optim mode, the expression is *not* removed. But the effect of printing an error message and aborting are. This is an artifact of the way the system assert is normally used and permits:

`ink_assert(!setsockopt(...));`

Allows the use of assertion in a plugin.

Note that when using the system “assert”, you do not have to worry about the condition as the code will be 'dead code eliminated' by the compiler. With `INKAssert` you do.

Example

```
switch (event) {
case EVENT_IMMEDIATE:
....
default:
INKAssert (!setsockopt(...));
break;
}
```

First release Traffic Server 5.2

INKReleaseAssert

Allows the use of assertion in a plugin**.**

Prototype `void INKReleaseAssert(expression);`

Arguments A boolean expression.

Description If expression is false, causes the Traffic Edge in debug AND optim mode to print the file name, line number and expression, then to abort.

Allows the use of assertion in a plugin.

First release Traffic Server 5.2

The INKfopen family

The `fopen` family of functions in C is normally used for reading configuration files, since `fgets` is an easy way to parse files on a line by line basis. The `INKfopen` family of functions is aimed at solving the same problem of buffered IO and line at a time IO in a platform independent manner. The `INKfopen` family of functions works exactly the same under Microsoft Windows NT as it does under any of the Unix platforms Traffic Edge runs on. Further, the `fopen` family of C library functions can only open a file if a file descriptor less than 256 is available. Traffic Edge often has more than 2000 file descriptors open at once, making the likelihood of an available file descriptor less than 256 very small. The INKfopen family can open files with descriptors greater than 256.

INKfopen not optimized for speed

The `INKfopen` family of routines is not intended for high speed IO or for flexibility. It is intended for reading and writing configuration information when corresponding usage of the `fopen` family of functions is inappropriate because of file descriptor and portability limitations. The INKfopen family of functions consists of:

INKfclose

Closes a file.

Prototype	`void` **`INKfclose`** `(INKFile` *`filep`*`)`
Arguments	*`filep`* is the file to close.
Description	Closes the file pointed to by *`filep`* and frees the data structures and buffers associated with it. If the file was opened for writing, any pending data is flushed.
Example	See the example for *INKfopen*.
First release	Traffic Server 3.0

INKfflush

Flushes a file.

Prototype	`void` **`INKfflush`** `(INKFile` *`filep`*`)`
Arguments	*`filep`* is the file to flush.
Description	Flushes pending data that has been buffered up in memory from previous calls to `INKfwrite`.
First release	Traffic Server 3.0

INKfgets

Reads a line from a file to a buffer.

Prototype	`char*` **`INKfgets`** `(INKFile` *`filep`*`, char *`*`buf`*`, int` *`length`*`)`
Arguments	*`filep`* is the file to read from. *`buf`* is the buffer to read into. *`length`* is the size of the buffer to read into.
Description	Reads a line from the file pointed to by *`filep`* into the buffer *`buf`*. Lines are terminated by a line feed character, '`\n`'. The line placed in the buffer includes the line feed character and is terminated with a `NUL`. If the line is longer than *`length`* bytes then only the first *`length`* `- 1` bytes are placed in *`buf`*.
First release	Traffic Server 3.0

INKfopen

Reads a line from a file to a buffer.

Prototype	`INKFile` **`INKfopen`** `(const char *`*`filename`*`, const char *`*`mode`*`)`
Arguments	*`filename`* is the name of the file to open. *`mode`* specifies whether to open the file for reading or writing. If *`mode`* is `“r”` then the file is opened for reading. `“w”`, then the file is opened for writing. `“a”` then the file is opened for appending. Currently `“r”` ,`“w”` and `“a”` are the only two valid modes for opening a file.

Description Opens a file for reading or writing and returns a descriptor for accessing the file. Descriptors of type `INKFile` can be greater than 256. `INKfopen` can open a file for reading or for writing, but not both. (This is a limitation of the current implementation).

Example The following example is taken from the append-transform plugin. The append-transform plugin appends text to the end of HTTP response bodies. This subroutine loads the text to be added from a file.

```
static int
load (const char *filename)
{
  INKFile fp;
  INKIOBufferBlock blk;
  INKIOBufferData data;
  char *p;
  int avail;
  int err;

  fp = INKfopen (filename, "r");
  if (!fp) {
    return 0;
  }

  append_buffer = INKIOBufferCreate ();
  append_buffer_reader = INKIOBufferReaderAlloc (append_buffer);

  for (;;) {
    blk = INKIOBufferStart (append_buffer);
    p = INKIOBufferBlockWriteStart (blk, &avail);

    err = INKfread (fp, p, avail);
    if (err > 0) {
      INKIOBufferProduce (append_buffer, err);
    } else {
      break;
    }
  }

  append_buffer_length = INKIOBufferReaderAvail (append_buffer_reader);

  INKfclose (fp);
  return 1;
}
```

First release Traffic Server 3.0

INKfread

Reads a specified number of bytes from a file to a buffer.

Prototype `int` **`INKfread`** `(INKFile` *`filep`*`, void *`*`buf`*`, int` *`length`*`)`

Arguments *`filep`* is the name of the file to read from.

`buf` is the buffer to read into.

`length` is the amount of data to read.

Description Attempts to read *`length`* bytes of data from the file pointed to by *`filep`* into the buffer *`buf`*. If the file was not opened for reading, `INKfread` returns `-1`. If an error occurs while reading the file, `INKfread` returns `-1`. If the end of the file is reached, `INKfread` returns `0`. Otherwise, `INKfread` returns the number of bytes read.

Example	See the example for *INKfopen*.
First release	Traffic Server 3.0

INKfwrite

Writes a specified number of bytes to a file.

Prototype	`int` **`INKfwrite`** `(INKFile` *`filep`*`, void *`*`buf`*`, int` *`length`*`)`
Arguments	*`filep`* is the file to write to. *`buf`* is the buffer containing the data to be written. *`length`* is the amount of data to write to *`filep`*.
Description	Attempts to write *`length`* bytes of data to the file pointed to by *`filep`* from the buffer *`buf`*. If the file was not opened for writing, `INKfwrite` returns `-1`. Otherwise, `INKfwrite` returns the number of bytes written. Unless an error occurs when writing data to the file, the number of bytes written is equal to *`length`*. One common error is an insufficient amount of space on disk.
First release	Traffic Server 3.0

Memory allocation

Traffic Edge provides five routines for allocating and freeing memory. These routines correspond to similar routines in the C library. For example, `INKrealloc` behaves like the C library routine `realloc`. There are two reasons to use the routines provided by Traffic Edge. The first is portability. The Traffic Edge API routines behave the same on all of Traffic Edge's supported platforms. For example, `realloc` does not accept an argument of `NULL` on some platforms. The second reason is that the Traffic Edge routines actually track the memory allocations by file and line number. This tracking is very efficient, is always turned on, and is useful for tracking down memory leaks.

The memory allocation functions are:

INKfree

Frees memory allocated by `INKmalloc` or `INKrealloc`.

Prototype	`void` **`INKfree`** `(void *`*`ptr`*`)`
Arguments	*`ptr`* is a pointer to the memory to deallocate.
Description	Releases the memory allocated by `INKmalloc` or `INKrealloc`. If *`ptr`* is `NULL`, `INKfree` does no operation.
First release	Traffic Server 3.0

INKmalloc

Allocates memory.

Prototype	`void*` **`INKmalloc`** `(unsigned int` *`size`*`)`
Arguments	*`size`* is the number of bytes to allocate.

Description Returns a pointer to *size* bytes of memory allocated from the heap. Traffic Edge uses INKmalloc internally for memory allocations. Always use INKfree to release memory allocated by INKmalloc; do not use free.

Returns A pointer to the newly allocated memory.

First release Traffic Server 3.0

INKrealloc

Changes the size of an allocated block of memory.

Prototype `void* INKrealloc (void *ptr, unsigned int size)`

Arguments *ptr* is the pointer to the memory to reallocate.
size is the number of bytes to allocate.

Description Changes the size of the memory block pointed to by *ptr* to *size* bytes and returns a pointer to the new block. It may not be possible to simply extend *ptr* to satisfy a request to increase the allocated block, so the returned pointer might point to a new block of memory. If *ptr* is NULL, INKrealloc behaves like INKmalloc and returns a pointer to the newly allocated memory.

Returns A pointer to the reallocated memory.

First release Traffic Server 3.0

INKstrdup

Returns a pointer to a duplicate string.

Prototype `char* INKstrdup (const char *str)`

Arguments *str* is a pointer to the null-terminated string to duplicate.

Description Returns a pointer to a new string that is a duplicate of the string pointed to by *str*. The memory for the new string is allocated using INKmalloc and should be freed by a call to INKfree.

Returns Pointer to the duplicated string.
Note: A valid null-terminated string may not be returned if the input *str* argument is not a valid pointer (i.e. a NULL argument would simply cause INKstrdup to return NULL).

First release Traffic Server 3.0

INKstrndup

Returns a pointer to a duplicate string of specified length.

Prototype `char* INKstrndup (const char *str, int length)`

Arguments *str* is a pointer to the string to duplicate.
length is the length of the string to duplicate.

Description Returns a pointer to a new string that is a duplicate of the string pointed to by *str* and *length* bytes long. The new string will be null-terminated. This API is very useful for transforming non-null terminated string values returned by APIs such as INKMimeHdrFieldStringValueGet into null-terminated string values. The memory for the new string is allocated using INKmalloc and should be freed by a call to INKfree.

Returns	Pointer to the duplicated string. Note: A valid null-terminated string may not be returned if the input *str* argument is not a valid pointer (i.e. a `NULL` argument would simply cause `INKstrndup` to return `NULL`).
First release	Traffic Server 3.0

Thread functions

The Traffic Edge API thread functions enable you to create, destroy, and identify threads within Traffic Edge. Multithreading enables a single program to have more than one stream of execution and to process more than one transaction at a time.

Threads serialize their access to shared resources and data using the `INKMutex` type, described in *Mutexes, on page 101*.

The thread functions are:

INKThreadCreate

Creates a new thread.

Prototype	`INKThread` **`INKThreadCreate`** `(INKThreadFunc` *`func`*`, void *`*`data`*`)`
Arguments	`INKThreadFunc` *`func`* is the function that the new thread executes. `void *`*`data`* is the data passed as an argument to *`func`*.
Description	Creates a new thread and calls *`func`* with the argument *`data`*. When *`func`* exits, the thread is destroyed automatically. Note: the `INKThread` return pointer does not provide any indication of the status of the new thread, and cannot be modified.
Returns	A valid pointer to an `INKThread` object if successful. A `NULL` pointer in case of an error.
First release	Traffic Server 3.0

INKThreadDestroy

Destroys a thread.

Prototype	`INKReturnCode` **`INKThreadDestroy`** `(INKThread` *`thread`*`)`
Description	Destroys a thread and frees all memory and associated data structures. This should only be called on threads that have been initialized using `INKThreadInit`.
Returns	`INK_SUCCESS` if successful. `INK_ERROR` if an error occurs.
First release	Traffic Server 3.0

INKThreadInit

Initializes a thread.

Prototype	`INKThread INKThreadInit (void)`
Description	Initializes a thread for use by Traffic Edge. This function should only be used if you create your own thread using something other than the `INKThreadCreate` function. This should not be called more than once for any given thread.
Returns	A valid pointer to an `INKThread` object if successful. A `NULL` pointer in case of an error.
First release	Traffic Server 3.0

INKThreadSelf

Obtain a thread identifier.

Prototype	`INKThread INKThreadSelf (void)`
Description	Returns the thread identifier for the currently executing thread.
Returns	A valid pointer to an `INKThread` object if successful. A `NULL` pointer in case of an error.
First release	Traffic Server 3.0

HTTP functions

Hook functions

INKHttpHookAdd

Adds an HTTP hook.

Prototype	`INKReturnCode` **`INKHttpHookAdd`** `(INKHttpHookId` *`id`*`, INKCont` *`contp`*`)`
Description	Adds *`contp`* to the end of the list of global HTTP hooks specified by *`id`*. Since `INKHttpHookAdd` is adding *`contp`* to a global list this function is only safe to call from the plugin initialization routine.
Returns	`INK_SUCCESS` if the hook is successfully added. `INK_ERROR` if the hook **is not** added.
First release	Traffic Server 3.0

Session functions

IINKHttpSsnHookAdd

Adds an HTTP session hook.

Prototype

```
INKReturnCode INKHttpSsnHookAdd (INKHttpSsn ssnp, INKHttpHookID id,
          INKCont contp)
```

Description

Adds *contp* to the end of the list of HTTP transaction hooks specified by *id*. This means that *contp* is called back for every transaction within the session, at the point specified by the hook ID. Since *contp* is added to a session, it is not possible to call INKHttpSsnHookAdd from the plugin initialization routine; the plugin needs a handle to an HTTP session. See the following example.

Returns

INK_SUCCESS if the hook is successfully added.

INK_ERROR if the hook **is not** added.

First release Traffic Server 3.0

Example

```
#include InkAPI.h

static void txn_handler (INKHttpTxn txnp, INKCont contp)
{
     //handle transaction
}

static void handle_session (INKHttpSsn ssnp, INKCont contp)
{
     INKHttpSsnHookAdd (ssnp, INK_HTTP_TXN_START_HOOK, contp);
}

static int ssn_handler (INKCont contp, INKEvent event, void *edata)
{
    INKHttpSsn ssnp;
    INKHttpTxn txnp;

    switch (event){
    case INK_EVENT_HTTP_SSN_START:
       ssnp = (INKHttpSsn) edata;
       handle_session (ssnp, contp);
       INKHttpSsnReenable (ssnp, INK_EVENT_HTTP_CONTINUE);
       return 0;

    case INK_EVENT_HTTP_TXN_START:
       txnp = (INKHttpTxn) edata;
       txn_handler (txnp, contp);
       INKHttpTxnReenable (txnp, INK_EVENT_HTTP_CONTINUE);
       return 0;

    default:
         break;
    }
    return 0;
}

void INKPluginInit (int argc, const char *argv[])
{
    INKCont contp;
    contp = INKContCreate (ssn_handler, NULL);
    INKHttpHookAdd (INK_HTTP_SSN_START_HOOK, contp);
}
```

INKHttpSsnReenable

Re-enables an HTTP session.

Prototype `INKReturnCode` **`INKHttpSsnReenable`** `(INKHttpSsn` *`ssnp`*`, INKEvent` *`event`*`)`

Description Notifies the HTTP session *`ssnp`* that the plugin is done processing the current hook. If `INK_EVENT_HTTP_CONTINUE` is specified for *`event`*, then the plugin wants the session to continue. If `INK_EVENT_HTTP_ERROR` is specified for *`event`*, then the plugin wants the session to be terminated and for an error to be sent back to the client if no response has already been sent.

Returns `INK_SUCCESS` if the session is successfully re-enabled.

`INK_ERROR` if the hook **is not** added.

First release Traffic Server 3.5

HTTP transaction functions

INKHttpTxnCacheLookupStatusGet

Stores the current cache lookup status for the ongoing transaction. Also stores the number of cache lookup operations already performed.

Prototype `INKReturnCode INKHttpTxnCacheLookupStatusGet (INKHttpTxn txnp, int *lookup_status)`

Arguments `INKHttpTxn txnp` is the ongoing transaction.

`int *lookup_status` is set to the lookup status.

Description Obtains the status of the current cache lookup for the ongoing transaction `txnp` in the `lookup_status` variable.

This function should only be called from `INK_HTTP_CACHE_LOOKUP_COMPLETE_HOOK`.

The possible status values returned in `lookup_status` are:

`INK_CACHE_LOOKUP_MISS` - Document was not in the cache. It will be fetched from the OS.

`INK_CACHE_LOOKUP_HIT_STALE` - Document was present in the cache but stale. A fresher version will be fetched from the OS (IMS request).

`INK_CACHE_LOOKUP_HIT_FRESH` - Document was present in the cache and is fresh. Document will be served from the cache.

`INK_CACHE_LOOKUP_SKIPPED` - Traffic Edge didn't perform a cache lookup as the request was not cacheable (url looks dynamic or request marked as noncacheable).

Returns `INK_SUCCESS` if the API is called successfully.

`INK_ERROR` if an error occurs while calling the API or if an argument is invalid.

First release Traffic Server 5.2

INKHttpTxnCachedReqGet

Gets the cached request header for a specified HTTP transaction.

Prototype `INKReturnCode **INKHttpTxnCachedReqGet** (INKHttpTxn txnp, INKMBuffer *bufp, INKMLoc *hdr_loc)`

Description Retrieves the cached request header from the HTTP transaction `txnp` and stores the cached request header in `bufp`, at location `hdr_loc`.

Call after READ_CACHE_HDR_HOOK.

Caution: Do not modify any cached request headers returned by `INKHttpTxnCachedReqGet`. The underlying data structure is read-only.

Release the returned `hdr_loc` with a call to `INKHandleMLocRelease`.

Returns If the cached request header does not exist, then `INKHttpTxnCachedReqGet` returns `0`.

Otherwise returns `1`.

First release Traffic Server 3.0

INKHttpTxnCachedRespGet

Gets the cached response header for a specified HTTP transaction.

Prototype	`int` **`INKHttpTxnCachedRespGet`** `(INKHttpTxn` *`txnp`*`, INKMBuffer *`*`bufp`*`, INKMLoc *`*`hdr_loc`*`)`
Description	Retrieves the cached response header from the HTTP transaction *`txnp`* and stores the cached response header in *`bufp`*, at location *`hdr_loc`*. Call after SEND_RESPONSE_HDR_HOOK. Caution: Do not modify any cached response headers returned by `INKHttpTxnCachedRespGet`. The underlying data structure is read-only. Release the returned *`hdr_loc`* with a call to `INKHandleMLocRelease`.
Returns	If the cached response header does not exist, then `INKHttpTxnCachedRespGet` returns `0`. Otherwise returns `1`.
First release	Traffic Server 3.0

INKHttpTxnClientIncomingPortGet

Gets the port on which the incoming request is received.

Prototype	`int` **`INKHttpTxnClientIncomingPortGet`** `(INKHttpTxn` *`txnp`*`)`
Description	Returns the port on which the HTTP transaction *`txnp`* was received. This is not the destination port in the URL. It is the proxy port to which the client browser is pointed. Call after TXN_START_HOOK.
Returns	The port number in host byte order. Returns -1 if an error occurred.
First release	Traffic Server 3.5

INKHttpTxnClientIPGet

Gets the client IP address for a specified HTTP transaction.

Prototype	`unsigned int` **`INKHttpTxnClientIPGet`** `(INKHttpTxn` *`txnp`*`)`
Description	Returns the IP address of the client for the HTTP transaction *`txnp`*. `INKHttpTxnClientIPGet` returns the IP address in network byte order. Call after TXN_START_HOOK.
Returns	The client IP address. Returns 0 if an error occurred.
First release	Traffic Server 3.0

INKHttpTxnClientRemotePortGet

Gets the remote host's port number for a specified HTTP transaction.

Prototype	`INKReturnCode INKHttpTxnClientRemotePortGet(INKHttpTxn txnp, int *port)`
Arguments	`INKHttpTxn txnp` is an HTTP transaction. `int *port` is set to the client's remote port value (port number used by the client when creating a socket connection with the proxy for the transaction `txnp`) in network byte order.
Description	Obtains the port number of the remote host for the specified HTTP transaction. The port number is returned in network byte order. **Note**: this is an exception to the rule that port numbers are retrieved in host byte order. The proxy port on which the connection was accepted can be retrieved using `INKHttpTxnClientIncomingPortGet`.
Returns	`INK_SUCCESS` if the API is called successfully. `INK_ERROR` if an error occurs while calling the API or if an argument is invalid.
First release	Traffic Server 5.2

NKHttpTxnClientReqGet

Gets the client request header for a specified HTTP transaction.

Prototype	`int` **`INKHttpTxnClientReqGet`** `(INKHttpTxn txnp, INKMBuffer *bufp, INKMLoc *hdr_loc)`
Description	Retrieves the client request header from the HTTP transaction *txnp*. `INKHttpTxnClientReqGet` stores the client request header in *bufp*, at location *hdr_loc*. Call after READ_REQUEST_HDR_HOOK. Release the returned *hdr_loc* with a call to `INKHandleMLocRelease`.
Returns	If the client request header does not exist or in case of an error, then `INKHttpTxnClientReqGet` returns `0`. Otherwise returns `1`.
First release	Traffic Server 3.0

INKHttpTxnClientRespGet

Gets the client response header for a specified HTTP transaction.

Prototype	`int` **`INKHttpTxnClientRespGet`** `(INKHttpTxn txnp, INKMBuffer *bufp, INKMLoc *hdr_loc)`
Description	Retrieves the client response header from the HTTP transaction *txnp*. `INKHttpTxnClientRespGet` stores the client response header in *bufp*, at location *hdr_loc*. Call after SEND_RESPONSE_HOOK. Release the returned *hdr_loc* with a call to `INKHandleMLocRelease`.

Returns If the client response header does not exist or in the case of an error, then `INKHttpTxnClientRespGet` returns `0`.

Otherwise returns `1`.

First release Traffic Server 3.0

INKHttpTxnErrorBodySet

Sets the format and content of the error body (or response data) that Traffic Edge sends to clients.

Prototype
```
INKReturnCode INKHttpTxnErrorBodySet (INKHttpTxn txnp, char *buf,
        int buflength, char *mimetype)
```

Arguments *txnp* is the HTTP transaction to act upon.

buf contains the error (or response) body. The error body can be text, an HTML document, image, or another format. Before you call `INKHttpTxnErrorBodySet`, be sure to allocate *buf* using `INKmalloc`.

buflength is the length of the error body.

mimetype contains the format of the error body. If you want to set the *mimetype* to a value other than `NULL`, you must allocate *mimetype* using `INKmalloc` before you call `INKHttpTxnErrorBodySet`.

Description Sets the format of the error body that Traffic Edge sends back when sending an error or response to a client. The error body data is stored in the buffer *buf*. If the error body is just plain text, setting *mimetype* to `NULL` works fine. If the error body is HTML then *mimetype* should be `"text/html"`. If the error body is a JPEG image then *mimetype* should be `"image/jpeg"`.

Note: Traffic Edge automatically calls `INKfree` to free *buf* when *buf* is no longer needed; make sure that the buffer *buf* is allocated by a call to `INKmalloc`. Similarly, if you want to set *mimetype* to something other than `NULL`, make sure that you allocate *mimetype* with a call to `INKmalloc`. Traffic Edge automatically calls `INKfree` to free *mimetype*.

Call after SEND_RESPONSE_HDR_HOOK.

Returns `INK_SUCCESS` if the operation completes successfully.

`INK_ERROR` if an error occurs.

First release Traffic Server 3.0

INKHttpTxnHookAdd

Adds a continuation to the list of HTTP transaction hooks for a specified HTTP transaction.

Prototype
```
INKReturnCode INKHttpTxnHookAdd (INKHttpTxn txnp, INKHttpHookID id,
        INKCont contp)
```

Description Adds *contp* to the end of the list of HTTP transaction hooks specified by *id*. Since *contp* is added to a transaction, it is not possible to call `INKHttpTxnHookAdd` from the plugin initialization routine but only when the plugin has a handle to an HTTP transaction.

Call after HTTP_TXN_START_HOOK.

Returns `INK_SUCCESS` if the operation completes successfully.

`INK_ERROR` if an error occurs.

First release Traffic Server 3.0

INKHttpTxnNextHopIPGet

Gets the IP address of the next server from which Traffic Edge tries to retrieve requested HTTP content.

Prototype	`unsigned int` **`INKHttpTxnNextHopIPGet`** `(INKHttpTxn` *`txnp`*`)`
Description	Returns the IP address of the next server from which Traffic Edge attempts to retrieve the requested document, in network byte order. This IP address could be the origin server IP address or it could be the parent proxy's IP address. Call after SEND_REQUEST_HDR_HOOK.
Returns	Returns the IP address of the next server from which Traffic Edge attempts to retrieve the request, in network byte order. Returns 0 if an error occurred.
First release	Traffic Server 3.0

INKHttpTxnParentProxyGet

Gets the parent proxy name and port, if parent proxying is enabled.

Prototype	`INKReturnCode` **`INKHttpTxnParentProxyGet`** `(INKHttpTxn` *`txnp`*`,` `char **`*hostname`*`, int *`*`port`*`)`
Description	Retrieves the value set previously by `INKHttpParentProxySet`. Does not return values set in `records.config` parameter `proxy.config.http.parent_proxies` or in `parent.config` file. This function can be called from within any txn hook. The `hostname` string returned **must not** be deallocated. **Note**: if parent proxying is not enabled, `INKHttpTxnParentProxyGet` returns `NULL` in *`hostname`* and `-1` in *`port`*.
Returns	`INK_SUCCESS` if the operation completes successfully. `INK_ERROR` if an error occurs.
First release	Traffic Server 3.0

INKHttpTxnParentProxySet

Sets the parent proxy name and port.

Prototype	`INKReturnCode` **`INKHttpTxnParentProxySet`** `(INKHttpTxn` *`txnp`*`,` `char *`*`hostname`*`, int` *`port`*`)`
Description	This can be used to overwrite the value set in `records.config` parameter `proxy.config.http.parent_proxies` or in `parent.config` file. Call before or within CACHE_LOOKUP_COMPLETE.
Returns	`INK_SUCCESS` if the operation completes successfully. `INK_ERROR` if an error occurs.
First release	Traffic Server 3.0

INKHttpTxnReenable

Tells a transaction whether or not the processing of a particular hook has completed.

Prototype `INKReturnCode` **`INKHttpTxnReenable`** `(INKHttpTxn` *`txnp`*`, INKEvent` *`event`*`)`

Description Notifies the HTTP transaction *`txnp`* that the plugin is done processing the current hook. If `INK_EVENT_HTTP_CONTINUE` is specified for *`event`*, then the plugin wants the transaction to continue. If `INK_EVENT_HTTP_ERROR` is specified for *`event`*, then the plugin wants the transaction to be terminated and for an error to be sent back to the client if no response has already been sent.

You must always re-enable the HTTP transaction after the processing of each transaction event. However, never re-enable twice. Re-enabling twice is a serious error.

When *`event`* is set to `INK_EVENT_HTTP_ERROR`, Traffic Edge performs different processing depending on the type of hook involved.

`INK_HTTP_TXN_START_HOOK`: The transaction is stopped right away, the connection to the client is closed, and no response is sent back to the origin server.

`INK_HTTP_READ_REQUEST_HDR_HOOK`: Traffic Edge does not send any request to the origin server, it directly sends a 500 to the client.

`INK_HTTP_SEND_REQUEST_HDR_HOOK`: Traffic Edge opens a connection to the origin server, sends an empty request to the origin server, and sends back 500 to the client. Then the connection to the origin server is closed.

`INK_HTTP_READ_RESPONSE_HDR_HOOK, INK_HTTP_SEND_RESPONSE_HOOK, INK_HTTP_OS_DNS_HOOK, INK_HTTP_READ_CACHE_HDR_HOOK, and INK_HTTP_CACHE_LOOKUP_COMPLETE_HOOK`: Traffic Edge receives all the headers of the response from the origin server, then closes the connection to the origin server and sends a 500 to the client. TS does not receive the response body.

`INK_HTTP_TXN_CLOSE_HOOK`: The client receives whatever answer was sent by the origin server because with this hook, the response has already been sent to the client.

Returns `INK_SUCCESS` if the operation completes successfully.

`INK_ERROR` if an error occurs.

First release Traffic Server 3.0

INKHttpTxnServerIPGet

Gets the origin server IP address for a specified HTTP transaction.

Prototype `unsigned int` **`INKHttpTxnServerIPGet`** `(INKHttpTxn` *`txnp`*`)`

Description Returns the IP address of the origin server specified by the client request in network byte order. `INKHttpTxnServerIPGet` returns `0` if it is called before `INK_HTTP_OS_DNS_HOOK` in a transaction.

Call after `INK_HTTP_OS_DNS_HOOK`.

Returns Returns the origin server IP address in network byte order.

Returns 0 if an error occurred.

First release Traffic Server 3.0

INKHttpTxnServerReqGet

Gets the server request header from a specified HTTP transaction.

Prototype	`int` **`INKHttpTxnServerReqGet`** `(INKHttpTxn` *`txnp`*`, INKMBuffer *`bufp`*`, INKMLoc *`hdr_loc`*`)`
Description	Retrieves the server request header from the HTTP transaction *`txnp`*. `INKHttpTxnServerReqGet` stores the server request header in *`bufp`*, at location *`hdr_loc`*. Call after SEND_REQUEST_HDR_HOOK. Release the returned *`hdr_loc`* with a call to `INKHandleMLocRelease`.
Returns	If the server request header does not exist or in the case of an error, then `INKHttpTxnServerReqGet` returns `0`. Otherwise returns `1`.
First release	Traffic Server 3.0

INKHttpTxnServerRespGet

Gets the server response header from a specified HTTP transaction.

Prototype	`int` **`INKHttpTxnServerRespGet`** `(INKHttpTxn` *`txnp`*`, INKMBuffer *`bufp`*`, INKMLoc *`hdr_loc`*`)`
Description	Retrieves the server response header from the HTTP transaction *`txnp`*. `INKHttpTxnServerRespGet` stores the server response header in *`bufp`*, at location *`hdr_loc`*. Call after READ_RESPONSE_HDR_HOOK. Release the returned *`hdr_loc`* with a call to `INKHandleMLocRelease`.
Returns	If the server response header does not exist or in the case of an error, then `INKHttpTxnServerRespGet` returns `0`. Otherwise returns `1`.
First release	Traffic Server 3.0

INKHttpTxnSsnGet

Returns the session handle associated to a specified HTTP transaction.

Prototype	`INKHttpSsn` **`INKHttpTxnSsnGet`** `(INKHttpTxn` *`txnp`*`)`
Description	Retrieves the `INKHttpSsn` handle associated with the HTTP transaction *`txnp`*. Call after TXN_START_HOOK.
Returns	The session handle associated with the specified HTTP transaction. `INK_ERROR_PTR` if error.
First release	Traffic Server 3.0

INKHttpTxnTransformedRespCache

Indicates whether or not Traffic Edge writes transformed documents to the cache.

Prototype	`INKReturnCode` **`INKHttpTxnTransformedRespCache`** `(INKHttpTxn` *`txnp`*`, int` *`on`*`)`
Description	Specifies whether the transformed document should be written to the cache or not. If a transformation is occurring the default is for the transformed copy to be written to the cache. The default maintains a rule that only a single version of a document will be written to the cache for a single transaction. It is valid for that rule to be broken by specifying that both the transformed and the un-transformed documents be written to the cache. Calls need to be made prior to the actual transformation, (i.e. at the time of creating the transformation) rather than in the transformation. Note: This function does not overwrite HTTP directives, like Cache-Control or Expire, that determine whether or not a document may be cached. If the document can be cached, this function determines whether or not to cache the transformed version. Untransformed and transformed documents are cached as HTTP alternates. Call from within or after hook TXN_START_HOOK. If called after hook SEND_RESPONSE_HDR, this function will not be taken into account by TS.
Returns	`INK_SUCCESS` if the operation completes successfully. `INK_ERROR` if an error occurs.
First release	Traffic Server 3.0

INKHttpTxnTransformRespGet

Gets the transform response header from a specified HTTP transaction.

Prototype	`int` **`INKHttpTxnTransformRespGet`** `(INKHttpTxn` *`txnp`*`, INKMBuffer *`*`bufp`*`, INKMLoc *`*`offset`*`)`
Description	Retrieves the transform response header from the HTTP transaction *`txnp`* and stores the transform response header in *`bufp`*, at location *`offset`*. Call from within your transformation, before transform data is written to the downstream vconnection.
Returns	If the transform response header does not exist, then `INKHttpTxnTransformRespGet` returns `0`. Otherwise returns `1`.
First release	Traffic Server 3.0

INKHttpTxnUntransformedRespCache

Indicates whether or not Traffic Edge writes un-transformed documents to the cache.

Prototype `INKReturnCode` **`INKHttpTxnUntransformedRespCache`** `(INKHttpTxn` *`txnp`*`,`
`int` *`on`*`)`

Description Specifies whether the un-transformed document should be written to the cache or not. If there is no transformation occurring then the default is for the un-transformed copy to be written to the cache. If a transformation is occurring the default is for the un-transformed copy to not be written to the cache. The defaults maintain a rule that only a single version of a document will be written to the cache for a single transaction. It is valid for that rule to be broken by specifying that both the transformed and un-transformed document be written to the cache. Calls need to be made prior to the actual transformation, (i.e. at the time of creating the transformation) rather than in the transformation.

Note: This function does not overwrite HTTP directives, like Cache-Control or Expire, that determine whether or not a document can be cached. If the document can be cached, this function determines whether or not to cache the untransformed version. Untransformed and transformed documents are cached as HTTP alternates.

Call from within or after hook TXN_START_HOOK.

If called after hook SEND_RESPONSE_HDR, this function will not be taken into account by TS.

Returns `INK_SUCCESS` if the operation completes successfully.

`INK_ERROR` if an error occurs.

First release Traffic Server 3.0

Initiate Connection

INKHttpConnect

Sends an HTTP request through the Traffic Edge HTTP SM.

Prototype `InkReturnCode` **`INKHttpConnect`** `(unsigned int` *`ip`*`, int` *`port`*`, INKVConn *`vc`*`)`

Arguments `unsigned int` *`ip`* is the IP address used to set the value of the VC remote IP address. This is equivalent to a client IP address: IP from which the connection is originated. Value is in host byte order.

`int` *`port`* is the port used to set the value of the VC remote port. This is equivalent to a client port: port from which the connection is originated. Value is in host byte order.

`INKVConn` *`*vc`* is the VConnection returned.
Once VConnection is established, you can use regular VConnection operations (`INKVConnRead`, `INKVConnWrite`, etc).

Description	Sends an HTTP request through the Traffic Edge HTTP SM. The HTTP request goes through the Traffic Edge the same way a request from a client (for instance a browser) does. A typical scenario when using is: Call `INKHttpConnect`. Use `INKVConnWrite` to send an HTTP request. Use `INKVConnRead` to get the HTTP response. If needed, use `INKHttpParser` to parse the response. Note that the request and response go through the Traffic Edge HTTP SM. The request and the response can be cached and the transaction will be logged in squid.log. Also note that the ip address passed to `INKHttpConnect` will be used as the client IP address in `squid.log`.
Returns	`INK_SUCCESS` if API is called successfully. `INK_ERROR` if an error occurs while calling the API or if an argument is invalid.
First release	Traffic Server 5.2

Intercepting HTTP transaction functions

INKHttpTxnIntercept

Allows a plugin to intercept an HTTP client's request and to serve the content in place of the origin server.

Prototype	`INKReturnCode` **`INKHttpTxnIntercept`** `(INKCont` *`contp`*`, INKHttpTxn` *`txnp`*`)`
Arguments	`INKCont` *`contp`* is the continuation that is called to accept the connection. `INKHttpTxn` *`txnp`* is the current HTTP txn the plugin wants to intercept.

Description Allows a plugin to intercept an HTTP client's request and to serve the content in place of the origin server. The request is intercepted right after being read by Traffic Edge. The origin server is not contacted.

This API should be used in the `INK_HTTP_READ_REQUEST_HDR_HOOK` hook.

Once `INKHttpTxnIntercept` has been called, the handler of the continuation *`contp`* receives an event `INK_EVENT_NET_ACCEPT`. Note that the continuation passed should not have a `NULL` mutex or an error is returned.

The `void *`*`data`* passed to the handler of the continuation *`contp`* is a data of type NetVConnection representing the connection.

Once VConnection is established, user can use regular VConnection operations (`INKVConnRead`, `INKVConnWrite`, etc...).

A typical scenario when using `INKHttpTxnIntercept` is:

Call `INKHttpTxnIntercept` from hook `INK_HTTP_READ_REQUEST_HDR_HOOK`.

Get called back on the continuation's handler passed as argument to `INKHttpTxnIntercept`.

Get the VC from argument `void *`*`data`* .

Use `INKVConnRead` to get the HTTP request. Note that you will not receive an event `INK_VCONN_READ_COMPLETE`, only `INK_VCONN_READ_READY`, as the number of characters to read is unknown. You should rely on `INKTHttpParser` to parse the request and return a status `INK_PARSE_DONE` when request is fully received (escape sequence “\r\n\r\n” read).

Use `INKHttpParser` to parse the request.

Use `INKVConnWrite` to write the HTTP response.

Note: the request and response do not go through the Traffic Edge HTTP state machine. So the request and response are not cached by Traffic Edge. The request is logged in `squid.log`.

Returns `INK_SUCCESS` if the API is called successfully.

`INK_ERROR` if an error occurs while calling the API or if an argument is invalid. This error is also returned if the continuation passed has a `NULL` mutex.

First release Traffic Server 5.2

INKHttpTxnServerIntercept

Allows a plugin to intercept an HTTP request sent to an origin server and to serve the content in place of the origin server.

Prototype `INKReturnCode` **`INKHttpTxnServerIntercept`** `(INKCont` *`contp,`* `INKHttpTxn` *`txnp`* `)`

Arguments `INKCont` *`contp`* is the continuation that is called to accept the connection.

`INKHttpTxn` *`txnp`* is the current HTTP txn the plugin wants to intercept.

Description Allows a plugin to intercept an HTTP request sent to an origin server and to serve the content in place of the origin server. The origin server is not contacted.

This API should be used in the `INK_HTTP_READ_REQUEST_HDR_HOOK` hook.

Once `INKHttpTxnServerIntercept` has been called, the handler of the continuation *`contp`* receives an event `INK_EVENT_NET_ACCEPT`. Note that the continuation passed should not have a `NULL` mutex or an error is returned.

The `void *`*`data`* passed to the handler of the continuation *`contp`* is a data of type NetVConnection representing the connection.

Once VConnection is established, you can use regular VConnection operations (`INKVConnRead`, `INKVConnWrite`, etc...).

A typical scenario when using `INKHttpTxnServerIntercept` is:

Call `INKHttpTxnServerIntercept` from hook `INK_HTTP_READ_REQUEST_HDR_HOOK` .

Get called back on the continuation's handler passed as argument to `INKHttpTxnServerIntercept`.

Get the VC from argument `void *`*`data`*.

Use `INKVConnRead` to get the HTTP header. Note that you will not receive an event `INK_VCONN_READ_COMPLETE`, only `INK_VCONN_READ_READY`, as the number of characters to read is unknown. You should rely on `INKTHttpParser` to parse the request and return a status `INK_PARSE_DONE` when request is fully received (escape sequence “\r\n\r\n” read).

Use `INKHttpParser` to parse the request.

Use `INKVConnWrite` to write the HTTP response.

Note that the request and response go through the Traffic Edge HTTP SM. The request and response can be cached. The request is logged in `squid.log`.

Returns `INK_SUCCESS` if the API is called successfully.

`INK_ERROR` if an error occurs while calling the API or if an argument is invalid. This error is also returned if the continuation passed has a `NULL` mutex.

First release Traffic Server 5.2

Alternate selection functions

INKHttpAltInfoCachedReqGet

Gets the cached request header from the specified alternate information.

Prototype
```
INKReturnCode INKHttpAltInfoCachedReqGet (INKHttpAltInfo infop,
          INKMBuffer *bufp, INKMLoc *offset)
```

Description Retrieves the cached client request header from the alternate information*`infop`*.

Call from within HTTP_SELECT_ALT_HOOK.

Returns `INK_SUCCESS` if the operation completes successfully.

`INK_ERROR` if an error occurs.

First release Traffic Server 3.0

INKHttpAltInfoCachedRespGet

Gets the cached response header from the specified alternate information.

Prototype	`INKReturnCode` **`INKHttpAltInfoCachedRespGet`** `(INKHttpAltInfo` *`infop`*`, INKMBuffer *`*`bufp`*`, INKMLoc *`*`offset`*`)`
Description	Retrieves the cached client response header from the alternate information *`infop`*. Call from within HTTP_SELECT_ALT_HOOK.
Returns	`INK_SUCCESS` if the operation completes successfully. `INK_ERROR` if an error occurs.
First release	Traffic Server 3.0

INKHttpAltInfoClientReqGet

Gets the client request header from the specified alternate information.

Prototype	`INKReturnCode` **`INKHttpAltInfoClientReqGet`** `(INKHttpAltInfo` *`infop`*`, INKMBuffer *`*`bufp`*`, INKMLoc *`*`offset`*`)`
Description	Retrieves the client request header from the alternate information *`infop`*. Call from within HTTP_SELECT_ALT_HOOK.
Returns	`INK_SUCCESS` if the operation completes successfully. `INK_ERROR` if an error occurrs.
First release	Traffic Server 3.0

INKHttpAltInfoQualitySet

Sets the quality value for the specified alternate information.

Prototype	`INKReturnCode` **`INKHttpAltInfoQualitySet`** `(INKHttpAltInfo` *`infop`*`, float` *`quality`*`)`
Description	Sets the quality value for this alternate information *`infop`*. Call from within HTTP_SELECT_ALT_HOOK.
Returns	`INK_SUCCESS` if the operation completes successfully. `INK_ERROR` if an error occurs.
First release	Traffic Server 3.0

Handle release functions

INKHandleMLocRelease

Releases `INKMLoc` handles.

Prototype	`INKReturnCode` **`INKHandleMLocRelease`** `(INKMBuffer` *`bufp`*`, INKMLoc` *`parent`*`, INKMLoc` *`mloc`*`)`
Arguments	*`bufp`* is the marshal buffer containing the `INKMLoc` to be released. *`parent`* is the location of the parent object from which the handle was created. *`mloc`* is the `INKMLoc` to be released.
Description	Releases the `INKMLoc mloc` created from the `INKMLoc parent`. If there is no parent `INKMLoc`, use `INK_NULL_MLOC`. See *Release marshal buffer handles, on page 88* for a details about parent `INKMLocs` and the use of the null parent.
Returns	`INK_SUCCESS` if the handle is successfully released. `INK_ERROR` if the hook **is not** added.
First release	Traffic Server 3.5

INKHandleStringRelease

Releases string handles.

Prototype	`InkReturnCode` **`INKHandleStringRelease`** `(INKMBuffer` *`bufp`*`, INKMLoc` *`parent`*`, const char *`*`str`*`)`
Arguments	*`bufp`* is the marshal buffer containing the string to be released. *`parent`* is the location of the parent object from which the handle was created. *`str`* is the string to be released.
Description	Releases the string `str` created from the `INKMLoc` *`parent`*. Do not use `INKHandleStringRelease` for strings created by `INKUrlStringGet` (in that special case, use `INKfree`).
Returns	`INK_SUCCESS` if the string handle is successfully released. `INK_ERROR` if the hook **is not** added.
First release	Traffic Server 3.5

Marshal buffers

The marshal buffer or `INKMBuffer` is a heap data structure that stores parsed URLs, MIME headers and HTTP headers. You can allocate new objects out of marshal buffers, and change the values within the marshal buffer. Whenever you manipulate an object, you require the handle to the object (`INKMLoc`) and the marshal buffer containing the object (`INKMBuffer`).

Routines exist for manipulating the object based on these two pieces of information. See, for example:

✔*HTTP header functions, on page 168*

✔*URL functions, on page 178*

✔*MIME headers, on page 187*

The marshal buffer functions allow you to create and destroy Traffic Edge's marshal buffers, which are the data structures that hold parsed URLs, MIME headers, and HTTP headers.

Caution Any marshal buffer fetched by `INKHttpTxn*Get` (for example, `INKHttpTxnClientReqGet` or `INKHttpTxnServerRespGet`) will be used by other parts of the system. Be careful not to destroy these shared, transaction marshal buffers.

INKMBufferCreate

Creates a new marshal buffer.

Prototype	`INKMBuffer` **`INKMBufferCreate`** `(void)`
Description	Creates a new marshal buffer and initializes the reference count to 1.
Returns	A pointer to the new marshal buffer.
First release	Traffic Server 3.0

INKMBufferDestroy

Destroys a marshal buffer.

Prototype	`void` **`INKMBufferDestroy`** `(INKMBuffer` *`bufp`*`)`
Arguments	*`bufp`* is the marshal buffer to be destroyed.
Description	Ignores the reference count and destroys the marshal buffer *`bufp`*. The internal data buffer associated with the marshal buffer is also destroyed if the marshal buffer allocated it.
First release	Traffic Server 3.0

HTTP header functions

The HTTP header functions are:

INKHttpHdrClone

Copies an HTTP header to a marshal buffer and returns the `INKMLoc` location of the copied header.

Prototype	`INKMLoc` **`INKHttpHdrClone`** `(INKMBuffer` *`dest_bufp`*`, INKMBuffer` *`src_bufp`*`, INKMLoc` *`src_hdr`*`)`
Description	Copies the contents of the HTTP header located at **`src_hdr`** within the marshal buffer *`src_bufp`* to the marshal buffer located at **`dest_bufp`**. If the HTTP header located at the **`src_hdr`** is a HTTP request header, ensure that it has a valid method, url, protocol and version. If the HTTP header located at the **`src_hdr`** is a HTTP response header, ensure that it has a valid protocol, version, status and reason. Call after READ_REQUEST_HDR_HOOK, if it is a transaction header. Release the returned handle with a call to `INKHandleMLocRelease`.
Returns	Returns the `INKMLoc` location of the copied header. `INK_ERROR_PTR` if error.
First release	Traffic Server 3.5

INKHttpHdrCopy

Copies an HTTP header.

Prototype	`INKReturnCode` **`INKHttpHdrCopy`** `(INKMBuffer` *`dest_bufp`*`, INKMLoc` *`dest_hdr_loc`*`, INKMBuffer` *`src_bufp`*`, INKMLoc` *`src_hdr_loc`*`)`
Description	Copies the contents of the HTTP header located at *`src_hdr_loc`* within the marshal buffer *`src_bufp`* to the HTTP header located at *`dest_hdr_loc`* within the marshal buffer *`dest_bufp`*. `INKHttpHdrCopy` works correctly even if *`src_bufp`* and *`dest_bufp`* point to different marshal buffers. Make sure that the destination HTTP header exists (has been created) before copying into it. `INKHttpHdrCopy` automatically makes sure that types of the source and destination HTTP headers match; if the destination type is not equal to the source type, `INKHttpHdrCopy` calls `INKHttpHdrTypeSet`. Do not call `INKHttpHdrTypeSet` on the destination header after using `INKHttpHdrCopy`. Call after READ_REQUEST_HDR_HOOK, if it is a transaction header. **Note**: `INKHttpHdrCopy` appends the port number to the domain of the URL portion of the header. For example, `http://www.inktomi.com` appears as: `http://www.inktomi.com:80/` in the destination buffer.
Returns	`INK_SUCCESS` if successful. `INK_ERROR` if an error occurs.
First release	Traffic Server 3.0

INKHttpHdrCreate

Creates a new HTTP header.

Prototype	`INKMLoc` **`INKHttpHdrCreate`** `(INKMBuffer` *`bufp`*`)`
Description	Creates a new HTTP header with the marshal buffer *`bufp`*. When newly created, the HTTP header is assigned an `INKHttpType` value of `INK_HTTP_TYPE_UNKNOWN`. You can change the type after creating the header using `INKHttpHdrTypeSet`, but you can only change the type once. You cannot modify the type after setting it. Release with a call to `INKHandleMLocRelease`.
Returns	A pointer to the new HTTP header.
First release	Traffic Server 3.0

INKHttpHdrDestroy

Destroys an HTTP header.

Prototype	`INKReturnCode` **`INKHttpHdrDestroy`** `(INKMBuffer` *`bufp`*`, INKMLoc` *`hdr_loc`*`)`
Description	Destroys the HTTP header located at *`hdr_loc`* within the marshal buffer *`bufp`*. **Caution**: Do not forget to use `INKHandleMLocRelease` to release the handle *`hdr_loc`*.
Returns	`INK_SUCCESS` if the operation completes successfully. `INK_ERROR_PTR` if error.
First release	Traffic Server 3.0

INKHttpHdrLengthGet

Calculates the length of an HTTP header.

Prototype `int` **`INKHttpHdrLengthGet`** `(INKMBuffer` *`bufp`*`, INKMLoc` *`hdr_loc`*`)`

Description Calculates the length of the HTTP header located at *`hdr_loc`* within the marshal buffer *`bufp`* if it were returned as a string. This is the length of the HTTP header in its un-parsed form and is also the number of bytes that will be added to the IO buffer by a call to `INKHttpHdrPrint`.

The header could be a request header, response header, or a standalone header that you have created. Be sure to call this function appropriately (if you want the length of a request header, call this function after READ_REQUEST_HDR_HOOK, for example).

Returns The length of the specified HTTP header.

`INK_ERROR` if error.

First release Traffic Server 3.0

INKHttpHdrMethodGet

Gets the method portion of an HTTP request header.

Prototype `const char*` **`INKHttpHdrMethodGet`** `(INKMBuffer` *`bufp`*`, INKMLoc` *`hdr_loc`*`, int *`*`length`*`)`

Description Retrieves the method from the HTTP header located at *`hdr_loc`* within the marshal buffer *`bufp`*. The length of the returned string is placed in the *`length`* argument. If *`length`* is NULL, then no attempt is made to de-reference it.

It is an error to try and retrieve the method from an HTTP header which is not of type `INK_HTTP_TYPE_REQUEST`.

Call after READ_REQUEST_HDR_HOOK, if it is a transaction header.

Release with a call to `INKHandleStringRelease`.

Returns A pointer to the method portion of the specified HTTP request header.

`INK_ERROR_PTR` if error.

First release Traffic Server 3.0

INKHttpHdrMethodSet

Set the HTTP method.

Prototype `INKReturnCode` **`INKHttpHdrMethodSet`** `(INKMBuffer` *`bufp`*`, INKMLoc` *`hdr_loc`*`, const char *`*`value`*`, int` *`length`*`)`

Description Sets the method in the HTTP header located at *`hdr_loc`* within the marshal buffer *`bufp`*. If *`length`* is `-1` then it is assumed that *`value`* is null-terminated. Otherwise, the length of the string *`value`* is taken to be *`length`*. The string is copied to within *`bufp`*, so it is okay to modify or delete *`value`* after calling `INKHttpHdrMethodSet`. It is an error to try and set the method in an HTTP header which is not of type `INK_HTTP_TYPE_REQUEST`.

Call after READ_REQUEST_HDR_HOOK, if it is a transaction header.

Returns `INK_SUCCESS` if successful.

`INK_ERROR` if an error occurs.

First release Traffic Server 3.0

INKHttpHdrPrint

Prints the HTTP header to an IO buffer.

Prototype	`INKReturnCode` **`INKHttpHdrPrint`** `(INKMBuffer` *`bufp`*`, INKMLoc` *`hdr_loc`*`, INKIOBuffer` *`iobufp`*`)`
Description	Formats the HTTP header located at *`hdr_loc`* within the marshal buffer *`bufp`* into the IO buffer *`iobufp`*. See *IO buffers, on page 128* for information on allocating an IO Buffer and retrieving data from within one.
Returns	`INK_SUCCESS` if the operation completes successfully. `INK_ERROR` if an error occurs.
First release	Traffic Server 3.0

INKHttpHdrReasonGet

Gets the reason phrase from an HTTP header.

Prototype	`const char*` **`INKHttpHdrReasonGet`** `(INKMBuffer` *`bufp`*`, INKMLoc` *`hdr_loc`*`, int *`*`length`*`)`
Description	Retrieves the reason phrase from the HTTP header located at *`hdr_loc`* within the marshal buffer *`bufp`*. The length of the returned string is placed in the *`length`* argument. It is an error to try and retrieve the reason phrase from an HTTP header which is not of type `INK_HTTP_TYPE_RESPONSE`. Call after READ_RESPONSE_HDR_HOOK, if it is a transaction header. **Note**: the returned string is **not** guaranteed to be null-terminated. Release with a call to `INKHandleStringRelease`.
Returns	Pointer to the reason phrase in the specified HTTP header. `INK_ERROR_PTR` if error.
First release	Traffic Server 3.0

INKHttpHdrReasonLookup

Provides the default reason phrase for a specified Traffic Edge HTTP status code.

Prototype	`const char*` **`INKHttpHdrReasonLookup`** `(INKHttpStatus` *`status`*`)`
Description	Returns the default reason phrase for the status code *`status`*. `INKHttpHdrReasonLookup` returns a string which **is** null-terminated and **should not** be freed or released. It's a global shared value.
Returns	Pointer to the default reason phrase for the specified Traffic Edge status code. `INK_ERROR_PTR` if error.
First release	Traffic Server 3.0

INKHttpHdrReasonSet

Sets the reason phrase in an HTTP header.

Prototype	`INKReturnCode` **`INKHttpHdrReasonSet`** `(INKMBuffer` *`bufp`*`, INKMLoc` *`hdr_loc`*`, const char *`*`value`*`, int` *`length`*`)`
Description	Sets the reason phrase in the HTTP header located at *`hdr_loc`* within the marshal buffer *`bufp`*. If *`length`* is `-1` then it is assumed that *`value`* is null-terminated. Otherwise, the length of the string *`value`* is taken to be *`length`*. The string is copied to within *`bufp`*, so it is okay to modify or delete *`value`* after calling `INKHttpHdrReasonSet`. It is an error to try and set the reason phrase in an HTTP header which is not of type `INK_HTTP_TYPE_RESPONSE`. Call after READ_RESPONSE_HDR_HOOK, if it is a transaction header.
Returns	`INK_SUCCESS` if the operation completes successfully. `INK_ERROR` if the operation **does not** complete successfully.
First release	Traffic Server 3.0

INKHttpHdrStatusGet

Retrieves the status code from an HTTP header.

Prototype	`INKHttpStatus` **`INKHttpHdrStatusGet`** `(INKMBuffer` *`bufp`*`, INKMLoc` *`hdr_loc`*`)`
Description	Retrieves the status code from the HTTP header located at *`hdr_loc`* within the marshal buffer *`bufp`*. It is an error to try and retrieve the status code from an HTTP header which is not of type `INK_HTTP_TYPE_RESPONSE`. `INKHttpStatus` is an enumerated type. Call after READ_RESPONSE_HDR_HOOK, if it is a transaction header.
Returns	The status code from the specified HTTP header. `INK_ERROR` if error.

Example The values of `INKHttpStatus` are the following:

```
typedef enum
{
    INK_HTTP_STATUS_NONE                            = 0,

    INK_HTTP_STATUS_CONTINUE                        = 100,
    INK_HTTP_STATUS_SWITCHING_PROTOCOL              = 101,

    INK_HTTP_STATUS_OK                              = 200,
    INK_HTTP_STATUS_CREATED                         = 201,
    INK_HTTP_STATUS_ACCEPTED                        = 202,
    INK_HTTP_STATUS_NON_AUTHORITATIVE_INFORMATION = 203,
    INK_HTTP_STATUS_NO_CONTENT                      = 204,
    INK_HTTP_STATUS_RESET_CONTENT                   = 205,
    INK_HTTP_STATUS_PARTIAL_CONTENT                 = 206,

    INK_HTTP_STATUS_MULTIPLE_CHOICES                = 300,
    INK_HTTP_STATUS_MOVED_PERMANENTLY               = 301,
    INK_HTTP_STATUS_MOVED_TEMPORARILY               = 302,
    INK_HTTP_STATUS_SEE_OTHER                       = 303,
    INK_HTTP_STATUS_NOT_MODIFIED                    = 304,
    INK_HTTP_STATUS_USE_PROXY                       = 305,

    INK_HTTP_STATUS_BAD_REQUEST                     = 400,
    INK_HTTP_STATUS_UNAUTHORIZED                    = 401,
    INK_HTTP_STATUS_PAYMENT_REQUIRED                = 402,
    INK_HTTP_STATUS_FORBIDDEN                       = 403,
    INK_HTTP_STATUS_NOT_FOUND                       = 404,
    INK_HTTP_STATUS_METHOD_NOT_ALLOWED              = 405,
    INK_HTTP_STATUS_NOT_ACCEPTABLE                  = 406,
    INK_HTTP_STATUS_PROXY_AUTHENTICATION_REQUIRED = 407,
    INK_HTTP_STATUS_REQUEST_TIMEOUT                 = 408,
    INK_HTTP_STATUS_CONFLICT                        = 409,
    INK_HTTP_STATUS_GONE                            = 410,
    INK_HTTP_STATUS_LENGTH_REQUIRED                 = 411,
    INK_HTTP_STATUS_PRECONDITION_FAILED             = 412,
    INK_HTTP_STATUS_REQUEST_ENTITY_TOO_LARGE        = 413,
    INK_HTTP_STATUS_REQUEST_URI_TOO_LONG            = 414,
    INK_HTTP_STATUS_UNSUPPORTED_MEDIA_TYPE          = 415,

    INK_HTTP_STATUS_INTERNAL_SERVER_ERROR           = 500,
    INK_HTTP_STATUS_NOT_IMPLEMENTED                 = 501,
    INK_HTTP_STATUS_BAD_GATEWAY                     = 502,
    INK_HTTP_STATUS_SERVICE_UNAVAILABLE             = 503,
    INK_HTTP_STATUS_GATEWAY_TIMEOUT                 = 504,
    INK_HTTP_STATUS_HTTPVER_NOT_SUPPORTED           = 505
} INKHttpStatus;
```

First release Traffic Server 3.0

INKHttpHdrStatusSet

Sets the status code within an HTTP header.

Prototype `INKReturnCode` **`INKHttpHdrStatusSet`** `(INKMBuffer` *`bufp`*`, INKMLoc` *`hdr_loc`*`, INKHttpStatus` *`status`*`)`

Description Sets the status code in the HTTP header located at *`hdr_loc`* within the marshal buffer *`bufp`*. It is an error to try and set the status code in an HTTP header which is not of type `INK_HTTP_TYPE_RESPONSE`.

Call after READ_RESPONSE_HDR_HOOK, if it is a transaction header.

Returns `INK_SUCCESS` if successful.

`INK_ERROR` if an error occurs.

First release Traffic Server 3.0

INKHttpHdrTypeGet

Retrieves the HTTP header type.

Prototype `INKHttpType` **`INKHttpHdrTypeGet`** `(INKMBuffer` *`bufp`*`, INKMLoc` *`hdr_loc`*`)`

Description Retrieves the type of the HTTP header located at *`hdr_loc`* within the marshal buffer *`bufp`*. `INKHttpType` is an enumerated type.

```
typedef enum
{
    INK_HTTP_TYPE_UNKNOWN,
    INK_HTTP_TYPE_REQUEST,
    INK_HTTP_TYPE_RESPONSE
} INKHttpType;
```

Returns The type of the specified HTTP header.

`INK_ERROR` if error.

First release Traffic Server 3.0

INKHttpHdrTypeSet

Sets the HTTP header type.

Prototype `INKReturnCode` **`INKHttpHdrTypeSet`** `(INKMBuffer` *`bufp`*`, INKMLoc` *`hdr_loc`*`, INKHttpType` *`type`*`)`

Description Sets the type of the HTTP header located at *`hdr_loc`* within the marshal buffer *`bufp`* to *`type`*. Use `INKHttpHdrTypeSet` only after you create an HTTP header. The `INKHttpHdrCreate` function automatically assigns the new header a type of `INK_HTTP_TYPE_UNKNOWN`, and you would only use `INKHttpHdrTypeSet` to change the type of a header from `INK_HTTP_TYPE_UNKNOWN` to either `INK_HTTP_TYPE_REQUEST` or `INK_HTTP_TYPE_RESPONSE`. You can only change the type once. You cannot modify the type after setting it.

Returns INK_SUCCESS if successful.

INK_ERROR if an error occurs.

First release Traffic Server 3.0

INKHttpHdrUrlGet

Gets the location of the URL portion of an HTTP header.

Prototype INKMLoc **INKHttpHdrUrlGet** (INKMBuffer *bufp*, INKMLoc *req_hdr_loc*)

Description Retrieves the URL from the HTTP header located at *req_hdr_loc* within the marshal buffer *bufp*. It is an error to try and retrieve the URL from an HTTP header which is not of type INK_HTTP_TYPE_REQUEST.

Call after READ_REQUEST_HDR_HOOK, if it is a transaction header.

Release with a call to INKHandleMLocRelease. When you release the handle created by INKHttpHdrUrlGet, the parent should be *req_hdr_loc*.

Returns The URL from the specified HTTP header.

INK_ERROR_PTR if error.

First release Traffic Server 3.0

INKHttpHdrUrlSet

Sets a URL location within an HTTP request header.

Prototype INKReturnCode **INKHttpHdrUrlSet** (INKMBuffer *bufp*, INKMLoc *hdr_loc*, INKMLoc *url*)

Description Sets the URL in the HTTP header located at *hdr_loc* within the marshal buffer *bufp*. It is an error to try and set the URL in an HTTP header which is not of type INK_HTTP_TYPE_REQUEST.

Call after READ_REQUEST_HDR_HOOK, if it is a transaction header.

Returns INK_SUCCESS if successful.

INK_ERROR if an error occurs.

First release Traffic Server 3.0

INKHttpHdrVersionGet

Retrieves the HTTP version of the specified HTTP header.

Prototype int **INKHttpHdrVersionGet** (INKMBuffer *bufp*, INKMLoc *hdr_loc*)

Description Retrieves the version from the HTTP header located at *hdr_loc* within the marshal buffer *bufp*. An HTTP version is composed of a major and a minor version. Traffic Edge encodes the major version in the upper 16 bits of the returned integer and the minor version in the lower 16 bits. The macros INK_HTTP_MAJOR (ver) and INK_HTTP_MINOR (ver) can be used to extract the major and minor versions respectively.

Call after READ_REQUEST_HDR_HOOK, if it is a transaction header.

Returns The HTTP version from the specified HTTP header.
`INK_ERROR` if error.

First release Traffic Server 3.0

INKHttpHdrVersionSet

Sets the HTTP version of the specified HTTP header.

Prototype `INKReturnCode` **`INKHttpHdrVersionSet`** `(INKMBuffer` *`bufp`*`, INKMLoc` *`hdr_loc`*`, int` *`ver`*`)`

Description Sets the version in the HTTP header located at *`hdr_loc`* within the marshal buffer *`bufp`* to *`ver`*. An HTTP version is composed of a major and a minor version. Traffic Edge encodes the major version in the upper 16 bits of the returned integer and the minor version in the lower 16 bits. The macro `INK_HTTP_VERSION (maj, min)` can be used to encode a major and minor version into the single integer expected by `INKHttpHdrVersionSet`.
Call after READ_REQUEST_HDR_HOOK, if it is a transaction header.

Returns `INK_SUCCESS` if the operation completes successfully.
`INK_ERROR` if the operation **does not** complete successfully.

First release Traffic Server 3.0

INKHttpParserClear

Clears an HTTP parser.

Prototype `INKReturnCode` **`INKHttpParserClear`** `(INKHttpParser` *`parser`*`)`

Description Clears the specified HTTP *`parser`* so it may be used again.
Call after READ_REQUEST_HDR_HOOK, if it is a transaction header.

Returns `INK_SUCCESS` if successful.
`INK_ERROR` if an error occurs.

First release Traffic Server 3.0

INKHttpParserCreate

Creates a parser for HTTP headers.

Prototype `INKHttpParser` **`INKHttpParserCreate`** `(void)`

Description Creates an HTTP parser. The parser's data structure contains information about the header being parsed. A single HTTP parser can be used multiple times, though not simultaneously. Before being used again, the parser must be cleared by calling `INKHttpParserClear.`

Returns Parser structure for HTTP headers.
`INK_ERROR_PTR` if error.

First release Traffic Server 3.0

INKHttpParserDestroy

Destroys an HTTP parser.

Prototype	`INKReturnCode` **`INKHttpParserDestroy`** `(INKHttpParser` *`parser`*`)`
Description	Destroys the specified HTTP *parser* and frees the associated memory.
Returns	`INK_SUCCESS` if the operation completes successfully. `INK_ERROR` if the operation **does not** complete successfully.
First release	Traffic Server 3.0

INKHttpHdrParseReq

Parses an HTTP request header.

Prototype	`int` **`INKHttpHdrParseReq`** `(INKHttpParser` *`parser`*`, INKMBuffer` *`bufp`*`, INKMLoc` *`hdr_loc`*`, const char **`*start`*`, const char *`*`end`*`)`
Description	Parses an HTTP request header. The HTTP header *`hdr_loc`* must already be created, and must reside inside the marshal buffer *`bufp`*. The *`start`* argument points to the current position of the string buffer being parsed and the *`end`* argument points to **one byte after** the end of the buffer to be parsed. On return, *`start`* is modified to point past the last character parsed. It is possible to parse an HTTP request header a single byte at a time using repeated calls to `INKHttpHdrParseReq`. As long as an error does not occur, the `INKHttpHdrParseReq` function will consume that single byte and ask for more. Call after READ_REQUEST_HDR_HOOK, if it is a transaction header.
Returns	`INK_PARSE_ERROR` is returned on error. `INK_PARSE_DONE` is returned when a `\r\n\r\n` pattern is encountered, indicating the end of the header. `INK_PARSE_CONT` is returned if parsing of the header stopped because the end of the buffer was reached.
First release	Traffic Server 3.0

INKHttpHdrParseResp

Parses an HTTP response header.

Prototype	`int` **`INKHttpHdrParseResp`** `(INKHttpParser` *`parser`*`, INKMBuffer` *`bufp`*`, INKMLoc` *`hdr_loc`*`, const char **`*start`*`, const char *`*`end`*`)`
Description	Parses an HTTP response header. The HTTP header *`hdr_loc`* must already be created, and must reside inside the marshal buffer *`bufp`*. The *`start`* argument points to the current position of the string buffer being parsed and the *`end`* argument points to **one byte after** the end of the buffer to be parsed. On return, *`start`* is modified to point past the last character parsed. It is possible to parse an HTTP response header a single byte at a time using repeated calls to `INKHttpHdrParseResp`. As long as an error does not occur, the `INKHttpHdrParseResp` function will consume that single byte and ask for more. Call after READ_RESPONSE_HDR_HOOK, if it is a transaction header.
Returns	`INK_PARSE_ERROR` is returned on error. `INK_PARSE_DONE` is returned when a `\r\n\r\n` pattern is encountered, indicating the end of the header. `INK_PARSE_CONT` is returned if parsing of the header stopped because the end of the buffer was reached
First release	Traffic Server 3.0

URL functions

The URL functions are:

INKUrlClone

Copies a URL from a specified location in a source marshal buffer to a target marshal buffer.

Prototype	`INKMLoc` **`INKUrlClone`** `(INKMBuffer` *`dest_bufp`*`, INKMBuffer` *`src_bufp`*`, INKMLoc` *`src_url_loc`*`)`
Arguments	*`src_bufp`* and *`dest_bufp`* are the source and destination marshal buffers. *`src_url_loc`* is the source URL location within the source marshal buffer.
Description	Copies the contents of the URL at location *`src_url_loc`* within the marshal buffer *`src_bufp`* to a location within the marshal buffer *`dest_bufp`*. `INKUrlClone`. Call after READ_REQUEST_HDR_HOOK, if it is in a transaction header. Release the returned handle with a call to `INKHandleMLocRelease`.
Returns	Returns the `INKMLoc` location of the copied URL. `INK_ERROR_PTR` if error.
First release	Traffic Server 3.5

INKUrlCopy

Copies a URL at a specified location in a source marshal buffer to a specified location in a target marshal buffer.

Prototype	`INKReturnCode` **`INKUrlCopy`** `(INKMBuffer` *`dest_bufp`*`, INKMLoc` *`dest_url_loc`*`, INKMBuffer` *`src_bufp`*`, INKMLoc` *`src_url_loc`*`)`
Arguments	*`src_bufp`* and *`dest_bufp`* are the source and destination marshal buffers. *`src_url_loc`* and *`dest_url_loc`* are the source and destination URL locations within the source and destination marshal buffers. The type `INKMLoc` is used for marshal buffer locations.
Description	Copies the contents of the URL at location *`src_url_loc`* within the marshal buffer *`src_bufp`* to the location *`dest_url_loc`* within the marshal buffer *`dest_bufp`*. `INKUrlCopy` works correctly even if *`src_bufp`* and *`dest_bufp`* point to different marshal buffers. It is important for the destination URL (its marshal buffer and `INKMLoc`) to have been created before copying into it. Call after READ_REQUEST_HDR_HOOK, if it is in a transaction header.
Returns	`INK_SUCCESS` if successful. `INK_ERROR` if an error occurs.
First release	Traffic Server 3.0

INKUrlCreate

Creates a new URL in a marshal buffer.

Prototype	`INKMLoc` **`INKUrlCreate`** `(INKMBuffer` *`bufp`*`)`
Description	Creates a new URL within the marshal buffer *`bufp`*. Release the resulting handle with a call to `INKHandleMLocRelease`, and destroy the URL with a call to `INKUrlDestroy` (note that if you destroy the URL, you must also release the handle). Call after READ_REQUEST_HDR_HOOK, if it is in a transaction header.

Returns	A location handle for the URL within the marshal buffer. `INK_ERROR_PTR` if error.
First release	Traffic Server 3.0

INKUrlDestroy

Destroys a specific URL within a marshal buffer.

Prototype	`INKReturnCode` **`INKUrlDestroy`** `(INKMBuffer` *`bufp`*`, INKMLoc` *`url_loc`*`)`
Description	Destroys the URL located at *`url_loc`* within the marshal buffer *`bufp`*. Call after READ_REQUEST_HDR_HOOK, if it is in a transaction header. **Caution**: Do not forget to release the handle *`url_loc`* with a call to `INKHandleMLocRelease`.
Returns	`INK_SUCCESS` if successful. `INK_ERROR` if an error occurs.
First release	Traffic Server 3.0

INKUrlPrint

Formats a URL stored in a marshal buffer to an INKIOBuffer.

Prototype	`INKReturnCode` **`INKUrlPrint`** `(INKMBuffer` *`bufp`*`, INKMLoc` *`url_loc`*`,` `INKIOBuffer` *`iobufp`*`)`
Description	Formats a URL stored in an `INKMBuffer` to an `INKIOBuffer`. Call after READ_REQUEST_HDR_HOOK, if it is in a transaction header.
Returns	`INK_SUCCESS` if successful. `INK_ERROR` if an error occurs.
First release	Traffic Server 3.5

INKUrlFtpTypeGet

Gets the FTP type of a specific URL.

Prototype	`int` **`INKUrlFtpTypeGet`** `(INKMBuffer` *`bufp`*`, INKMLoc` *`url_loc`*`)`
Description	Retrieves the FTP type portion of the URL located at *`url_loc`* within the marshal buffer *`bufp`*. Call after READ_REQUEST_HDR_HOOK, if it is within a transaction header.
Returns	Returns 65 if the FTP type is ASCII. Return 73 if the FTP type is binary. `INK_ERROR_PTR` if error.
First release	Traffic Server 3.0

INKUrlFtpTypeSet

Sets the FTP type of a specific URL.

Prototype	`INKReturnCode` **`INKUrlFtpTypeSet`** `(INKMBuffer` *`bufp`*`, INKMLoc` *`url_loc`*`, int` *`type`*`)`
Description	Sets the FTP type portion of the URL located at *`url_loc`* within the marshal buffer *`bufp`* to the value *`type`*. The valid values for the type argument are : *`0, 65('A'), 97('a'), 69('E'), 101('e'), 73 ('I')`* and *`105('i')`*. Call after READ_REQUEST_HDR_HOOK, if it is in a transaction header.
Returns	`INK_SUCCESS` if successful. `INK_ERROR` if an error occurs.
First release	Traffic Server 3.0

INKUrlHostGet

Gets the host portion of a specific URL.

Prototype	`const char*` **`INKUrlHostGet`** `(INKMBuffer` *`bufp`*`, INKMLoc` *`url_loc`*`, int *`*`length`*`)`
Description	Retrieves the host portion of the URL located at *`url_loc`* within the marshal buffer *`bufp`*. The length of the returned string is placed in the *`length`* argument. Call after READ_REQUEST_HDR_HOOK, if it is in a transaction header.
Returns	A pointer to the host portion of the specified URL. Release with a call to `INKHandleStringRelease`. `INK_ERROR_PTR` if error. Note: the returned string is **not** guaranteed to be null-terminated.
First release	Traffic Server 3.0

INKUrlHostSet

Sets the host portion of a URL to a specific value.

Prototype	`INKReturnCode` **`INKUrlHostSet`** `(INKMBuffer` *`bufp`*`, INKMLoc` *`url_loc`*`, const char *`*`value`*`, int` *`length`*`)`
Description	Sets the host portion of the URL located at *`url_loc`* within the marshal buffer *`bufp`* to the string *`value`*. If *`length`* is `-1` then `INKUrlHostSet` assumes that value is null-terminated. Otherwise, the length of the string *`value`* is taken to be *`length`*. The string is copied to within *`bufp`*, so it is okay to modify or delete *`value`* after calling `INKUrlHostSet`. Call after READ_REQUEST_HDR_HOOK, if it is in a transaction header.
Returns	`INK_SUCCESS` if successful. `INK_ERROR` if an error occurs.
First release	Traffic Server 3.0

INKUrlHttpFragmentGet

Gets a specified HTTP fragment of a URL.

Prototype	`const char* **INKUrlHttpFragmentGet** (INKMBuffer *bufp*, INKMLoc *url_loc*, int *length*)`
Description	Retrieves the HTTP fragment portion of the URL located at *url_loc* within the marshal buffer *bufp*. `INKUrlHttpFragmentGet` places the length of the returned string in the *length* argument. Call after READ_REQUEST_HDR_HOOK, if it is in a transaction header.
Returns	A pointer to the HTTP fragment portion of the specified URL. Release with a call to `INKHandleStringRelease`. `INK_ERROR_PTR` if error. Note: the returned string is **not** guaranteed to be null-terminated.
First release	Traffic Server 3.0

INKUrlHttpFragmentSet

Sets a specified HTTP fragment within a URL.

Prototype	`INKReturnCode **INKUrlHttpFragmentSet** (INKMBuffer *bufp*, INKMLoc *url_loc*, const char *value, int length)`
Description	Sets the HTTP fragment portion of the URL located at *url_loc* within the marshal buffer *bufp* to the string value. If *length* is `-1` then `INKUrlHttpFragmentSet` assumes that value is null-terminated. Otherwise, the length of the string value is taken to be *length*. The string is copied to within *bufp*, so it is okay to modify or delete value after calling `INKUrlHttpFragmentSet`. Call after READ_REQUEST_HDR_HOOK, if it is in a transaction header.
Returns	`INK_SUCCESS` if successful. `INK_ERROR` if an error occurs.
First release	Traffic Server 3.0

INKUrlHttpParamsGet

Gets the HTTP params portion of a specified URL.

Prototype	`const char* **INKUrlHttpParamsGet** (INKMBuffer *bufp*, INKMLoc *url_loc*, int *length)`
Description	Retrieves the HTTP params portion of the URL located at *url_loc* within the marshal buffer *bufp*. `INKUrlHttpParamsGet` places the length of the returned string in the *length* argument. Call after READ_REQUEST_HDR_HOOK, if it is in a transaction header.
Returns	A pointer to the HTTP params portion of the specified URL. Release with a call to `INKHandleStringRelease`. `INK_ERROR_PTR` if error. Note: the returned string is **not** guaranteed to be null-terminated.
First release	Traffic Server 3.0

INKUrlHttpParamsSet

Sets the HTTP params portion of a specified URL.

Prototype `INKReturnCode` **`INKUrlHttpParamsSet`** `(INKMBuffer` *`bufp`*`, INKMLoc` *`url_loc`*`, const char *`*`value`*`, int` *`length`*`)`

Description Sets the HTTP params portion of the URL located at *`url_loc`* within the marshal buffer *`bufp`* to the string *`value`*. If *`length`* is `-1` then `INKUrlHttpParamsSet` assumes that *`value`* is null-terminated. Otherwise, the length of the string *`value`* is taken to be *`length`*. `INKUrlHttpParamsSet` copies the string to within *`bufp`*, so it is okay to modify or delete *`value`* after calling `INKUrlHttpParamsSet`.

Call after READ_REQUEST_HDR_HOOK, if it is in a transaction header.

Returns `INK_SUCCESS` if successful.

`INK_ERROR` if an error occurs.

First release Traffic Server 3.0

INKUrlHttpQueryGet

Gets the HTTP query portion of a specified URL.

Prototype `const char*` **`INKUrlHttpQueryGet`** `(INKMBuffer` *`bufp`*`, INKMLoc` *`url_loc`*`, int *`*`length`*`)`

Description Retrieves the HTTP query portion of the URL located at *`url_loc`* within the marshal buffer *`bufp`*. `INKUrlHttpQueryGet` places the length of the returned string in the *`length`* argument.

Call after READ_REQUEST_HDR_HOOK, if it is in a transaction header.

Returns A pointer to the HTTP query portion of the specified URL. Release with a call to `INKHandleStringRelease`.

`INK_ERROR_PTR` if error.

Note: the returned string is **not** guaranteed to be null-terminated.

First release Traffic Server 3.0

INKUrlHttpQuerySet

Sets the HTTP query portion of a specified URL.

Prototype `INKReturnCode` **`INKUrlHttpQuerySet`** `(INKMBuffer` *`bufp`*`, INKMLoc` *`url_loc`*`, const char *`*`value`*`, int` *`length`*`)`

Description Sets the HTTP query portion of the URL located at *`url_loc`* within the marshal buffer *`bufp`* to the string *`value`*. If *`length`* is -1 then `INKUrlHttpQuerySet` assumes that *`value`* is null-terminated. Otherwise, the length of the string *`value`* is taken to be *`length`*. `INKUrlHttpQuerySet` copies the string to within *`bufp`*, so it is okay to modify or delete *`value`* after calling `INKUrlHttpQuerySet`.

Call after READ_REQUEST_HDR_HOOK, if it is in a transaction header.

Returns `INK_SUCCESS` if successful.

`INK_ERROR` if an error occurs.

First release Traffic Server 3.0

INKUrlLengthGet

Calculates the length of the string representation of a URL.

Prototype	`int` **`INKUrlLengthGet`** `(INKMBuffer` *`bufp`*`, INKMLoc` *`url_loc`*`)`
Description	Calculates the length of URL located at *`url_loc`* within the marshal buffer *`bufp`* if it were returned as a string. This length will be the same as the length returned by `INKUrlStringGet`. Call after READ_REQUEST_HDR_HOOK, if it is in a transaction header.
Returns	Returns the calculated length. `INK_ERROR` if error.
First release	Traffic Server 3.0

INKUrlParse

Parses the specified URL.

Prototype	`int` **`INKUrlParse`** `(INKMBuffer` *`bufp`*`, INKMLoc` *`url_loc`*`, const char **`*`start`*`, const char *`*`end`*`)`
Description	Parses a URL. The *`start`* pointer is both an input and an output parameter and marks the start of the URL to be parsed. After a successful parse, the start pointer equals the end pointer. The end pointer must be **one byte after** the last character you want to parse.The URL parsing routine assumes that everything between *`start`* and *`end`* is part of the URL. It is up to higher level parsing routines, such as `INKHttpHdrParseReq`, to determine the actual end of the URL. Call after READ_REQUEST_HDR_HOOK, if it is in a transaction header.
Returns	Returns `INK_PARSE_ERROR` if an error occurs, otherwise `INK_PARSE_DONE` is returned to indicate success.
First release	Traffic Server 3.0

INKUrlPasswordGet

Gets the password portion of a specified URL.

Prototype	`const char* ` **`INKUrlPasswordGet`** `(INKMBuffer` *`bufp`*`, INKMLoc` *`url_loc`*`, int *`*`length`*`)`
Description	Retrieves the password portion of the URL located at *`url_loc`* within the marshal buffer *`bufp`*. `INKUrlPasswordGet` places the length of the returned string in the *`length`* argument. Call after READ_REQUEST_HDR_HOOK, if it is in a transaction header.
Returns	A pointer to the password portion of the specified URL. Release with a call to `INKHandleStringRelease`. `INK_ERROR_PTR` if error. **Note**: the returned string is **not** guaranteed to be null-terminated.
First release	Traffic Server 3.0

INKUrlPasswordSet

Sets the password portion of a specified URL.

Prototype `INKReturnCode` **`INKUrlPasswordSet`** `(INKMBuffer` *`bufp`*`, INKMLoc` *`url_loc`*`, const char *`*`value`*`, int` *`length`*`)`

Description Sets the password portion of the URL located at *`url_loc`* within the marshal buffer *`bufp`* to the string *`value`*. If *`length`* is `-1` then `INKUrlPasswordSet` assumes that *`value`* is null-terminated. Otherwise, the length of the string *`value`* is taken to be *`length`*. `INKUrlPasswordSet` copies the string to within *`bufp`*, so it is okay to modify or delete *`value`* after calling `INKUrlPasswordSet`.

Call after READ_REQUEST_HDR_HOOK, if it is in a transaction header.

Returns `INK_SUCCESS` if successful.

`INK_ERROR` if an error occurs.

First release Traffic Server 3.0

INKUrlPathGet

Gets the path portion of a specified URL.

Prototype `const char*` **`INKUrlPathGet`** `(INKMBuffer` *`bufp`*`, INKMLoc` *`url_loc`*`, int *`*`length`*`)`

Description Retrieves the path portion of the URL located at *`url_loc`* within the marshal buffer *`bufp`*. `INKUrlPathGet` places the length of the returned string in the *`length`* argument.

Call after READ_REQUEST_HDR_HOOK, if it is in a transaction header.

Returns A pointer to the path portion of the specified URL. Release with a call to `INKHandleStringRelease`.

`INK_ERROR_PTR` if error.

Note: the returned string is **not** guaranteed to be null-terminated.

First release Traffic Server 3.0

INKUrlPathSet

Sets the path portion of a specified URL.

Prototype `INKReturnCode` **`INKUrlPathSet`** `(INKMBuffer` *`bufp`*`, INKMLoc` *`url_loc`*`, const char *`*`value`*`, int` *`length`*`)`

Description Sets the path portion of the URL located at *`url_loc`* within the marshal buffer *`bufp`* to the string *`value`*. If *`length`* is `-1` then `INKUrlPathSet` assumes that *`value`* is null-terminated. Otherwise, the length of the string *`value`* is taken to be *`length`*. `INKUrlPathSet` copies the string to within *`bufp`*, so it is okay to modify or delete *`value`* after calling `INKUrlPathSet`.

Call after READ_REQUEST_HDR_HOOK, if it is in a transaction header.

Returns `INK_SUCCESS` if successful.

`INK_ERROR` if an error occurs.

First release Traffic Server 3.0

INKUrlPortGet

Gets the port number portion of a specified URL.

Prototype	`int` **`INKUrlPortGet`** `(INKMBuffer` *`bufp`*`, INKMLoc` *`url_loc`*`)`
Description	Retrieves the port number portion of the URL located at *`url_loc`* within the marshal buffer *`bufp`*. Call after READ_REQUEST_HDR_HOOK, if it is in a transaction header.
Returns	The port number portion of the specified URL. `INK_ERROR` if error.
First release	Traffic Server 3.0

INKUrlPortSet

Sets the port number portion of a URL to a specified value.

Prototype	`INKReturnCode` **`INKUrlPortSet`** `(INKMBuffer` *`bufp`*`, INKMLoc` *`url_loc`*`, int` *`port`*`)`
Description	Sets the port number portion of the URL located at *`url_loc`* within the marshal buffer *`bufp`* to the value *`port`*. Call after READ_REQUEST_HDR_HOOK, if it is in a transaction header.
Returns	`INK_SUCCESS` if successful. `INK_ERROR` if an error occurs.
First release	Traffic Server 3.0

INKUrlSchemeGet

Gets the scheme portion of a specified URL.

Prototype	`const char*` **`INKUrlSchemeGet`** `(INKMBuffer` *`bufp`*`, INKMLoc` *`url_loc`*`, int` *`*length`*`)`
Description	Retrieves the scheme portion of the URL located at *`url_loc`* within the marshal buffer *`bufp`*. `INKUrlSchemeGet` places the length of the returned string in the *`length`* argument. Call after READ_REQUEST_HDR_HOOK, if it is in a transaction header.
Returns	A pointer to the scheme portion of the specified URL. Release with a call to `INKHandleStringRelease`. `INK_ERROR_PTR` if error. **Note**: the returned string is **not** guaranteed to be null-terminated.
First release	Traffic Server 3.0

INKUrlSchemeSet

Sets the scheme portion of a specified URL.

Prototype	`INKReturnCode` **`INKUrlSchemeSet`** `(INKMBuffer` *`bufp`*`, INKMLoc` *`url_loc`*`, const char *`*`value`*`, int` *`length`*`)`
Description	Sets the scheme portion of the URL located at *`url_loc`* within the marshal buffer *`bufp`* to the string *`value`*. If *`length`* is `-1` then `INKUrlSchemeSet` assumes that *`value`* is null-terminated. Otherwise, the length of the string *`value`* is taken to be *`length`*. `INKUrlSchemeSet` copies the string to within *`bufp`*, so it is okay to modify or delete *`value`* after calling `INKUrlSchemeSet`. Call after READ_REQUEST_HDR_HOOK, if it is in a transaction header.
Returns	`INK_SUCCESS` if successful. `INK_ERROR` if an error occurs.
First release	Traffic Server 3.0

INKUrlStringGet

Constructs a string representation of the URL located at *`url_loc`* within the marshal buffer *`bufp`*.

Prototype	`char*` **`INKUrlStringGet`** `(INKMBuffer` *`bufp`*`, INKMLoc` *`url_loc`*`, int *`*`length`*`)`
Description	Constructs a string representation of the URL located at *`url_loc`* within the marshal buffer *`bufp`*. `INKUrlStringGet` stores the length of the allocated string in the parameter *`length`*. This is the same length that `INKUrlLengthGet` returns. The returned string is allocated by a call to `INKmalloc`. It should be freed by a call to `INKfree`. If *`length`* is `NULL` then no attempt is made to de-reference it. Call after READ_REQUEST_HDR_HOOK, if it is in a transaction header.
Returns	A null-terminated string. `INK_ERROR_PTR` in case of an error.
First release	Traffic Server 3.0

INKUrlUserGet

Gets the user portion of a specified URL.

Prototype	`const char*` **`INKUrlUserGet`** `(INKMBuffer` *`bufp`*`, INKMLoc` *`url_loc`*`, int *`*`length`*`)`
Description	Retrieves the user portion of the URL located at *`url_loc`* within the marshal buffer *`bufp`*. `INKUrlUserGet` places the length of the returned string in the *`length`* argument. Call after READ_REQUEST_HDR_HOOK, if it is in a transaction header.
Returns	A pointer to the user portion of the specified URL. Release with a call to `INKHandleStringRelease`. `INK_ERROR_PTR` if error. **Note**: the returned string is **not** guaranteed to be null-terminated.
First release	Traffic Server 3.0

INKUrlUserSet

Sets the user portion of a specified URL.

Prototype	`INKReturnCode` **`INKUrlUserSet`** `(INKMBuffer` *`bufp`*`, INKMLoc` *`url_loc`*`, const char *`*`value`*`, int` *`length`*`)`
Description	Sets the user portion of the URL located at *`url_loc`* within the marshal buffer *`bufp`* to the string *`value`*. If *`length`* is -1 then `INKUrlUserSet` assumes that *`value`* is null-terminated. Otherwise, the length of the string *`value`* is taken to be *`length`*. `INKUrlUserSet` copies the string to within *`bufp`*, so it is okay to modify or delete *`value`* after calling `INKUrlUserSet`. Call after READ_REQUEST_HDR_HOOK, if it is in a transaction header.
Returns	`INK_SUCCESS` if successful. `INK_ERROR` if an error occurs.
First release	Traffic Server 3.0

MIME headers

MIME headers and fields can be components of request headers, response headers, or standalone headers created within your plugin. Make sure you call the MIME header functions appropriately; for example, if you want to clone a MIME header field within a request header, call `INKMimeHdrFieldClone` after `READ_REQUEST_HDR_HOOK`.

The MIME header functions are:

INKMimeHdrFieldAppend

Appends a field in a MIME header.

Prototype	`INKReturnCode` **`INKMimeHdrFieldAppend`** `(INKMBuffer` *`bufp`*`, INKMLoc` *`hdr_loc`*`, INKMLoc` *`field`*`)`
Description	Appends the MIME field located at *`field`* within the marshal buffer *`bufp`* into the MIME header located at *`hdr_loc`* within the marshal buffer *`bufp`*.
Returns	`INK_SUCCESS` if the API is called successfully. `INK_ERROR` if an error occurs while calling the API or if an argument is invalid.
First release	Traffic Server 3.0

INKMimeHdrFieldClone

Copies a MIME field to a marshal buffer, and returns the `INKMLoc` location of the copied field.

Prototype	`INKMLoc` **`INKMimeHdrFieldClone`** `(INKMBuffer` *`dest_bufp`*`, INKMLoc` *`dest_hdr`*`, INKMBuffer` *`src_bufp`*`, INKMLoc` *`src_hdr`*`, INKMLoc` *`src_field`*`)`
Description	Copies the contents of the MIME field located at *`src_field`* within the marshal buffer *`src_bufp`* to a MIME header located at *`dest_hdr`* within the marshal buffer *`dest_bufp`*.
Returns	The `INKMLoc` location of the copied field. Release the returned handle with a call to `INKHandleMLocRelease`. `INK_ERROR_PTR` if error.
First release	Traffic Server 3.5

INKMimeHdrFieldCopy

Copies a MIME field from a specified location to another specified location.

Prototype	`INKReturnCode` **`INKMimeHdrFieldCopy`** `(INKMBuffer` *`dest_bufp`*`, INKMLoc` *`dest_hdr`*`, INKMLoc` *`dest_field`*`, INKMBuffer` *`src_bufp`*`, INKMLoc` *`src_hdr`*`, INKMLoc` *`src_field`*`)`
Description	Copies the contents of the MIME field located at *`src_field`* within the marshal buffer *`src_bufp`* to the MIME field located at *`dest_field`* within the marshal buffer *`dest_bufp`*. `INKMimeHdrFieldCopy` works correctly even if *`src_bufp`* and *`dest_bufp`* point to different marshal buffers. **Note**: you must first create the destination MIME field before copying into it.
Returns	`INK_SUCCESS` if successful. `INK_ERROR` if an error occurs.
First release	Traffic Server 3.5

INKMimeHdrFieldCopyValues

Copies MIME field values from one location to another.

Prototype	`INKReturnCode` **`INKMimeHdrFieldCopyValues`** `(INKMBuffer` *`dest_bufp`*`, INKMLoc` *`dest_hdr`*`, INKMLoc` *`dest_field`*`, INKMBuffer` *`src_bufp`*`, INKMLoc` *`src_hdr`*`, INKMLoc` *`src_field`*`)`
Description	Copies the values contained within the MIME field located at *`src_field`* within the marshal buffer *`src_bufp`* to the MIME field located at *`dest_field`* within the marshal buffer *`dest_bufp`*. `INKMimeHdrFieldCopyValues` works correctly even if *`src_bufp`* and *`dest_bufp`* point to different marshal buffers. `INKMimeHdrFieldCopyValues` does not copy the field's name.
Returns	`INK_SUCCESS` if successful. `INK_ERROR` if an error occurs.
First release	Traffic Server 3.5

INKMimeHdrFieldCreate

Creates a new MIME field within a specified marshal buffer.

Prototype	`INKMLoc` **`INKMimeHdrFieldCreate`** `(INKMBuffer` *`bufp`*`, INKMLoc` *`hdr`*`)`
Description	Creates a new MIME field with the marshal buffer *`bufp`*.
Returns	The location of the new MIME field. Release with a call to `INKHandleMLocRelease`.
First release	Traffic Server 3.5

INKMimeHdrFieldDestroy

Deletes a specified MIME field from a marshal buffer.

Prototype	`void` **`INKMimeHdrFieldDestroy`** `(INKMBuffer` *`bufp`*`, INKMLoc` *`hdr`*`, INKMLoc` *`field`*`)`
Description	Destroys the MIME field located at *`field`* within the MIME header located at *`hdr`* within the marshal buffer *`bufp`*. After the call to `INKMimeHdrFieldDestroy`, you must release the `INKMLoc` handle *`field`* with a call to `INKHandleMLocRelease`.
First release	Traffic Server 3.5

INKMimeHdrFieldLengthGet

Calculates the length of a string representation of a specified MIME field.

Prototype	`int` **`INKMimeHdrFieldLengthGet`** `(INKMBuffer` *`bufp`*`, INKMLoc` *`hdr`*`, INKMLoc` *`field`*`)`
Description	Calculates the length of the MIME field located at *`field`* within the marshal buffer *`bufp`* if it were returned as a string. This is the length of the MIME field in its unparsed form.
Returns	The calculated length of a string representation of the specified MIME field. `INK_ERROR` if there is an error.
First release	Traffic Server 3.5

INKMimeHdrFieldNameGet

Gets the name and name length of a specified MIME field.

Prototype	`const char*` **`INKMimeHdrFieldNameGet`** `(INKMBuffer` *`bufp`*`, INKMLoc` *`hdr`*`, INKMLoc` *`field`*`, int *`*`length`*`)`
Description	Returns the name of the field located at *`field`* within the marshal buffer *`bufp`*. `INKMimeHdrFieldNameGet` places the length of the returned string in the *`length`* argument.
Returns	A pointer to the name of the specified field within the specified MIME header. Release the returned string with a call to `INKHandleStringRelease`. `INK_ERROR_PTR` if error. Note: the returned string is **not** guaranteed to be null-terminated.
First release	Traffic Server 3.5

INKMimeHdrFieldNameSet

Sets a specified MIME field's name.

Prototype `INKReturnCode` **`INKMimeHdrFieldNameSet`** `(INKMBuffer` *`bufp`*`, INKMLoc` *`hdr`*`, INKMLoc` *`field`*`, const char *`*`name`*`, int` *`length`*`)`

Description Sets the name of the field located at *`field`* within the marshal buffer *`bufp`* to the string *`name`*. If *`length`* is `-1` then `INKMimeHdrFieldNameSet` assumes that*`name`* is null-terminated. Otherwise, the length of the string *`name`* is taken to be *`length`*. `INKMimeHdrFieldNameSet` copies the string to within *`bufp`*, so it is okay to modify or delete *`name`* after calling `INKMimeHdrFieldNameSet`.

For `name`, use the `INK_MIME_FIELD_XXX` tokens when possible. See *Constant Index, on page 277*.

Returns `INK_SUCCESS` if successful.

`INK_ERROR` if an error occurs.

First release Traffic Server 3.5

INKMimeHdrFieldNext

Returns the next MIME field after a specified MIME field in a MIME header.

Prototype `INKMLoc` **`INKMimeHdrFieldNext`** `(INKMBuffer` *`bufp`*`, INKMLoc` *`hdr`*`, INKMLoc` *`field`*`)`

Description Conceptually, there are a list of MIME fields in a MIME header (see *Guide to Traffic Edge HTTP header system, on page 87*). `INKMimeHdrFieldNext` returns the location of the next field in the list after the field located at *`field`* within the marshal buffer *`bufp`*. If the next field is not found, a NULL pointer is returned.

Returns The location of the MIME field following the specified MIME field within the specified MIME header. Release the returned `INKMLoc` with a call to `INKHandleMLocRelease`. See the code example below.

`INK_ERROR_PTR` if error.

Example An example of a loop through each MIME field of an HTTP header:

```
field_loc = INKMimeHdrFieldGet (hdr_bufp, hdr_loc, 0);
  while (field_loc) {
       /* Temp variable used only for the loop */
       INKMLoc next_field_loc;

       /* Do your job with the field here */

       /* Get the next field and release the current one */
       next_field_loc = INKMimeHdrFieldNext (hdr_bufp, hdr_loc,
field_loc);
       INKHandleMLocRelease(hdr_bufp, hdr_loc, field_loc);
       field_loc = next_field_loc;
       }
```

First release Traffic Server 3.5

INKMimeHdrFieldNextDup

Returns the next duplicate MIME field after a specified MIME field in a MIME header.

Prototype `INKMLoc` **`INKMimeHdrFieldNextDup`** `(INKMBuffer` *`bufp`*`, INKMLoc` *`hdr`*`, INKMLoc` *`field`*`)`

Description MIME headers MAY contain more than one MIME field with the same name. Previous versions of Traffic Edge joined multiple fields with the same name into one field with composite values. This behavior comes at a performance cost, and causes inter-operability problems with some older clients and servers. Future versions of Traffic Edge will cease coalescing duplicate fields.

Your plugins should check for the presence of duplicate fields, and iterate over duplicate fields, by using `INKMimeHdrFieldNextDup`.`INKMimeHdrFieldNextDup` returns the location of the next duplicated field in the list after the field located at *`field`* within the marshal buffer *`bufp`*. If the next field is not found, a NULL pointer is returned.

Returns The location of the next duplicate MIME field that follows the specified field within the specified MIME header. Release with a call to `INKHandleMLocRelease`.

`INK_ERROR_PTR` if error.

First release Traffic Server 3.5

INKMimeHdrFieldValueAppend

Appends a string to a specified value in a MIME field.

Prototype `INKReturnCode` **`INKMimeHdrFieldValueAppend`** `(INKMBuffer` *`bufp`*`, INKMLoc` *`hdr`*`, INKMLoc` *`field`*`, int` *`idx`*`, const char *`*`value`*`, int` *`length`*`)`

Arguments *`bufp`* is the marshal buffer containing the MIME field.

`hdr` is the location of the parent object within the marshal buffer *`bufp`* from which *`field`* was retrieved.

`field` is the location of the MIME field to be appended to.

`idx` is the index of the field value to be appended. For example, in the MIME field `Foo: bar, car` the index of the value `bar` is `0`, and the index of `car` is `1`.

`value` is the string to be appended to the MIME field value at *`idx`*.

`length` is the length of the string *`value`* to be appended.

Description Appends the string stored in *`value`* to a specific value in the MIME field located at *`field`* within the marshal buffer *`bufp`*. The effect of `INKMimeHdrFieldValueAppend` is as if the previous value were retrieved, the string *`value`* were appended to it and this new string were stored back in the MIME field at the same position. The *`idx`* parameter specifies which value in the field to append to. If *`idx`* is not between `0` and `INKMimeHdrFieldValuesCount`(*`bufp`*, *`hdr, field`*) `- 1` then no operation will be performed.

Returns `INK_SUCCESS` if the string is successfully appended.

`INK_ERROR` if the hook **is not** added.

First release Traffic Server 3.5

INKMimeHdrFieldValueDateGet

Gets date value from a MIME field.

Prototype	`INKReturnCode` **`INKMimeHdrFieldValueDateGet`** `(INKMBuffer` *`bufp`*`, INKMLoc` *`hdr_loc`*`, INKMLoc` *`field`*`, time_t *`*`value`*`)`
Description	Retrieves a date value from within the MIME field located at *`field`* within the marshal buffer *`bufp`*. All values are stored as strings within the MIME field. `INKMimeHdrFieldValueDateGet` parses the string value to return an integer date representation.
Returns	The date value from the specified MIME header. `INK_SUCCESS` if the API is called successfully. `INK_ERROR` if an error occurs while calling the API or if an argument is invalid.
First release	Traffic Server 3.5

INKMimeHdrFieldValueDateInsert

Inserts a date value into a MIME field.

Prototype	`INKReturnCode` **`INKMimeHdrFieldValueDateInsert`** `(INKMBuffer` *`bufp`*`, INKMLoc` *`hdr_loc`*`, INKMLoc` *`field`*`, time_t` *`value`*`)`
Description	Inserts the data *`value`* into the MIME field located at *`field`* within the marshal buffer *`bufp`*. All values are stored as strings within the MIME field. `INKMimeHdrFieldValueDateInsert` simply formats the date into a string and then calls `INKMimeHdrFieldValueInsert`.
Returns	`INK_SUCCESS` if the API is called successfully. `INK_ERROR` if an error occurs while calling the API or if an argument is invalid.
First release	Traffic Server 3.5

INKMimeHdrFieldValueDateSet

Sets a date value in a MIME field.

Prototype	`INKReturnCode` **`INKMimeHdrFieldValueDateSet`** `(INKMBuffer` *`bufp`*`, INKMLoc` *`hdr_loc,`* `INKMLoc` *`field`*`, time_t` *`value`*`)`
Description	Sets a value in the MIME field located at *`field`* within the marshal buffer *`bufp`* to the date *`value`*. All values are stored as strings within the MIME field. `INKMimeHdrFieldValueDateSet` simply formats the date into a string and then calls `INKMimeHdrFieldValueStringSet`. This API has been deprecated by .
Returns	`INK_SUCCESS` if the API is called successfully. `INK_ERROR` if an error occurs while calling the API or if an argument is invalid.
First release	Traffic Server 3.5

INKMimeHdrFieldValueDelete

Deletes a specified value from a MIME field.

Prototype `INKReturnCode` **`INKMimeHdrFieldValueDelete`** `(INKMBuffer` *`bufp`*`, INKMLoc` *`hdr`*`, INKMLoc` *`field`*`, int` *`idx`*`)`

Description Removes and deletes a value from the MIME field located at *`field`* within the marshal buffer *`bufp`*. The *`idx`* parameter specifies which value should be deleted. If *`idx`* is not between `0` and `INKMimeHdrFieldValuesCount` (*`bufp`*, *`hdr, field`*) `- 1` then no operation will be performed.

Returns `INK_SUCCESS` if successful.

`INK_ERROR` if an error occurs.

First release Traffic Server 3.5

INKMimeHdrFieldValueIntGet

Gets an integer field value in a MIME field.

Prototype `INKReturnCode` **`INKMimeHdrFieldValueIntGet`** `(INKMBuffer` *`bufp`*`, INKMLoc` *`hdr_loc`*`, INKMLoc` *`field`*`, int` *`idx`*`, int *`value`*`)`

Description Retrieves an integer value from within the MIME field located at *`field`* within the marshal buffer *`bufp`*. The *`idx`* parameter specifies which value within the field to retrieve. The fields are numbered from `0` to `INKMimeHdrFieldValuesCount` (*`bufp`*, *`hdr, field`*) `- 1`. If *`idx`* does not lie within that range, `INKMimeHdrFieldValueIntGet` returns `(int) 0`. All values are stored as strings within the MIME field. `INKMimeHdrFieldValueIntGet` parses the string value to return an integer.

Returns The interger value from the specified MIME field.

`INK_SUCCESS` if the API is called successfully.

`INK_ERROR` if an error occurs while calling the API or if an argument is invalid.

First release Traffic Server 3.5

INKMimeHdrFieldValueIntInsert

Inserts an integer value into a MIME field.

Prototype `INKReturnCode` **`INKMimeHdrFieldValueIntInsert`** `(INKMBuffer` *`bufp`*`, INKMLoc` *`hdr_loc`*`, INKMLoc` *`field`*`, int` *`value`*`, int` *`idx`*`)`

Description Inserts the integer *`value`* into the MIME field located at *`field`* within the marshal buffer *`bufp`*. The *`idx`* parameter specifies where the inserted value should be put with respect to the other values already in the MIME field. If *`idx`* is `0` then the value is prepended to the list of values in the field. Increasing values of *`idx`* places the value further down the list of values. If *`idx`* is `-1` then the value is appended to the list of values. Normal usage is to specify `-1` for *`idx`* so that the value is appended to the list of values. All values are stored as strings within the MIME field. `INKMimeHdrFieldValueIntInsert` simply formats the integer into a string and then calls `INKMimeHdrFieldValueInsert`.

Returns `INK_SUCCESS` if the API is called successfully.

`INK_ERROR` if an error occurs while calling the API or if an argument is invalid.

First release Traffic Server 3.5

INKMimeHdrFieldValueIntSet

Sets an integer value within a MIME field.

Prototype `INKReturnCode` **`INKMimeHdrFieldValueIntSet`** `(INKMBuffer` *`bufp`*`, INKMLoc` *`hdr_loc`*`, INKMLoc` *`field`*`, int` *`idx`*`, int` *`value`*`)`

Description Sets a value in the MIME field located at *`field`* within the marshal buffer *`bufp`* to the integer *`value`*. The *`idx`* parameter specifies which value in the field to change. If *`idx`* is not between `0` and `INKMimeHdrFieldValuesCount` (*`bufp`*, *`hdr`*, *`field`*) `- 1` then no operation will be performed. All values are stored as strings within the MIME field. `INKMimeHdrFieldValueIntSet` simply formats the integer into a string and then calls `INKMimeHdrFieldValueSet`.

Returns `INK_SUCCESS` if the API is called successfully.

`INK_ERROR` if an error occurs while calling the API or if an argument is invalid.

First release Traffic Server 3.5

INKMimeHdrFieldValueStringGet

Gets a specified field value from a MIME header.

Prototype `INKReturnCode` **`INKMimeHdrFieldValueStringGet`** `(INKMBuffer` *`bufp`*`, INKMLoc` *`hdr_loc`*`, INKMLoc` *`field`*`, int` *`idx`*`, const char **`*value`*`, int *`*value_len`*`)`

Description Retrieves a string value from within the MIME field located at *`field`* within the marshal buffer *`bufp`*. The *`idx`* parameter specifies which field to retrieve. The fields are numbered from `0` to `INKMimeHdrFieldValuesCount` (*`bufp`*, *`hdr`*, *`field`*) `- 1`. If *`idx`* does not lie within that range then `NULL` will be returned. The length of the returned string is placed in the *`value_len`* argument. If *`value_len`* is `NULL` then no attempt is made to dereference it.

Returns A pointer to the specified field value in the MIME header. Release with a call to `INKHandleStringRelease`.

`INK_SUCCESS` if the API is called successfully.

`INK_ERROR` if an error occurs while calling the API or if an argument is invalid.

First release Traffic Server 3.5

INKMimeHdrFieldValueStringInsert

Inserts a value into a specified location within a MIME field.

Prototype `INKReturnCode` **`INKMimeHdrFieldValueStringInsert`** `(INKMBuffer` *`bufp`*`, INKMLoc` *`hdr_loc`*`, INKMLoc` *`field`*`, const char *`*value`*`, int` *`len`*`, int` *`idx`*`)`

Description Inserts the string *`value`* into the MIME field located at *`field`* within the marshal buffer *`bufp`*. If *`len`* is `-1` then `INKMimeHdrFieldValueStringInsert` assumes that *`value`* is null-terminated. Otherwise, the length of the string *`value`* is taken to be *`length`*. `INKMimeHdrFieldValueStringInsert` copies the string to within *`bufp`*, so it is okay to modify or delete *`value`* after calling `INKMimeHdrFieldValueStringSet`. The *`idx`* parameter specifies where the inserted value should be put with respect to the other values already in the MIME field. If *`idx`* is `0` then `INKMimeHdrFieldValueStringInsert` prepends the value to the list of values in the field. Increasing values of *`idx`* place the value further down the list of values. If *`idx`* is `-1`, `INKMimeHdrFieldValueStringInsert` appends the value to the list of values. Normal usage is to specify `-1` for *`idx`* so that the value is appended to the list of values.

Returns INK_SUCCESS if the API is called successfully.

INK_ERROR if an error occurs while calling the API or if an argument is invalid.

First release Traffic Server 3.5

INKMimeHdrFieldValueStringSet

Sets a value in a MIME field.

Prototype
```
INKReturnCode INKMimeHdrFieldValueStringSet (INKMBuffer bufp, INKMLoc
   hdr_loc, INKMLoc field, int idx, const char *value, int len)
```

Description Sets a value in the MIME field located at *field* within the marshal buffer *bufp* to the string *value*. If *len* is -1 then it is assumed that *value* is null-terminated. Otherwise, the length of the string *value* is taken to be *len*. The string is copied to within *bufp*, so it is okay to modify or delete *value* after calling INKMimeHdrFieldValueStringSet. The *idx* parameter specifies which value in the field to change. If *idx* is not between 0 and INKMimeHdrFieldValuesCount (*bufp*, *hdr, field*) - 1 then no operation will be performed. If *idx* is set to -1 then all the mime field values are returned. For instance, suppose the mime field is MyField: *value1, value2, value3*. If INKMimeHdrFieldGet is called with *idx* set to -1, it will return a pointer to "*value1, value2, value3*".

Note that like for other mime header manipulation APIs, the string is not null terminated.

First release Traffic Server 3.5

INKMimeHdrFieldValueUintGet

Gets unsigned integer field value in a MIME field.

Prototype
```
INKReturnCode INKMimeHdrFieldValueUintGet (INKMBuffer bufp, INKMLoc
   hdr_loc, INKMLoc field, int idx, unsigned int *value)
```

Description Retrieves an unsigned integer value from within the MIME field located at *field* within the marshal buffer *bufp*. The *idx* parameter specifies which field to retrieve. The fields are numbered from 0 to INKMimeHdrFieldValuesCount (*bufp*, *hdr, field*) - 1. If *idx* does not lie within that range, INKMimeHdrFieldValueGetUnit returns (unsigned int) 0. All values are stored as strings within the MIME field. INKMimeHdrFieldValueUintGet parses the string value to return an unsigned integer.

It is not possible to determine if INKMimeHdrFieldValueUintGet is returning an unsigned int value in error. If you need to check for errors in MIME header field values, you can fetch the header as a string and examine it. Here is some sample code that fetches MIME headers from marshal buffers into strings using INKMimeHdrFieldValueGet instead. The context of this example is that the plugin is processing an HTTP transaction and has access to a transaction.

Returns The unsigned integer value from the specified MIME field.

INK_SUCCESS if the API is called successfully.

INK_ERROR if an error occurs while calling the API or if an argument is invalid.

Example

```
static void
handle_string (INKHttpTxn txnp, INKCont contp) {
    INKMBuffer bufp;
    INKMLoc hdr_loc;
    INKMLoc field;
    int len;
    char* output_string;
    const char* value;
/* Fetch  the transaction's client request header into a marshal buffer.
*/
    if (!INKHttpTxnClientReqGet (txnp, &bufp, &hdr_loc)) {
        INKError ("couldn't retrieve client request header\n");
        goto done;
    }
    field=INKMimeHdrFieldFind(bufp, hdr_loc,
                                  INK_MIME_FIELD_CONTENT_LENGTH);

    if (!field) {
        INKError ("Content-Length field not found.\n");
        INKHandleMLocRelease (bufp, INK_NULL_MLOC, hdr_loc);
        goto done;
    }
    /* Obtain the value of the content length (normally an
     * unsigned int) as a string. */
    value=INKMimeHdrFieldValueGet (bufp, hdr_loc, field, 0, &len);

    if ((!value) || (len<=0))}
        INKHandleMLocRelease (bufp, hdr_loc, field);
        INKHandleMLocRelease (bufp, INK_NULL_MLOC, hdr_loc);
        goto done;
    }
   /* Allocate the string with an extra byte for the string terminator.
*/
    output_string = (char*) INKmalloc(len + 1);

    /* Copy the value. */
    strncpy (output_string, value, len);

    /* Terminate the string */
    output_string[len] = '\0';
/* Now that you have the MIME fields as a string, you can do
       whatever you want to do with it, for example, print it, or
       make sure it's an unsigned integer: either by using the
       atol C function or by scanning each ASCII character.  */
    INKDebug("my_plugin", "%s", output_string);

    /* Release handles and allocated memory. */
    INKHandleStringRelease (bufp, field, value);
    INKf... (... ... ...);
```

First release Traffic Server 3.5

INKMimeHdrFieldValueUIntInsert

Inserts an unsigned integer value into a MIME field.

Prototype `INKReturnCode` **`INKMimeHdrFieldValueUIntInsert`** (`INKMBuffer` *`bufp`*, `INKMLoc` *`hdr_loc`*, `INKMLoc` *`field`*, `unsigned int` *`value`*, `int` *`idx`*)

Description Inserts the unsigned integer *`value`* into the MIME field located at *`field`* within the marshal buffer *`bufp`*. The *`idx`* parameter specifies where the inserted value should be put with respect to the other values already in the MIME field. If *`idx`* is `0` then the value will be prepended to the list of values in the field. Increasing values of *`idx`* will place the value further down the list of values. If *`idx`* is `-1` then the value will be appended to the list of values. Normal usage is to specify `-1` for *`idx`* so that the value will be appended to the list of values. All values are stored as strings within the MIME field. `INKMimeHdrFieldValueUIntInsert` simply formats the unsigned integer into a string and then calls `INKMimeHdrFieldValueStringInsert`.

Returns `INK_SUCCESS` if the API is called successfully.

`INK_ERROR` if an error occurs while calling the API or if an argument is invalid.

First release Traffic Server 3.5

INKMimeHdrFieldValueUintSet

Sets a value in a MIME field to a specified unsigned integer.

Prototype `INKReturnCode` **`INKMimeHdrFieldValueUintSet`** (`INKMBuffer` *`bufp`*, `INKMLoc` *`hdr_loc`*, `INKMLoc` *`field`*, `int` *`idx`*, `unsigned int` *`value`*)

Description Sets a value in the MIME field located at *`field`* within the marshal buffer *`bufp`* to the unsigned integer *`value`*. The *`idx`* parameter specifies which value in the field to change. If *`idx`* is not between `0` and `INKMimeHdrFieldValuesCount`(bufp, *`hdr`*, *`field`*) `- 1` then no operation will be performed. All values are stored as strings within the MIME field. `INKMimeHdrFieldValueUintSet` simply formats the unsigned integer into a string and then calls `INKMimeHdrFieldValueStringSet`.

Returns `INK_SUCCESS` if the API is called successfully.

`INK_ERROR` if an error occurs while calling the API or if an argument is invalid.

First release Traffic Server 3.5

INKMimeHdrFieldValuesClear

Clears all values in a MIME field.

Prototype `INKReturnCode` **`INKMimeHdrFieldValuesClear`** (`INKMBuffer` *`bufp`*, `INKMLoc` *`hdr`*, `INKMLoc` *`field`*)

Description Removes and destroys all of the values within the MIME field located at *`field`* within the marshal buffer *`bufp`*.

Make sure you release any corresponding `INKMLoc` or string handles using `INKHandleMLocRelease` or `INKHandleStringRelease`.

Returns `INK_SUCCESS` if successful.

`INK_ERROR` if an error occurs.

First release Traffic Server 3.5

INKMimeHdrFieldValuesCount

Counts the values in a MIME field.

Prototype	`int` **`INKMimeHdrFieldValuesCount`** `(INKMBuffer` *`bufp`*`, INKMLoc` *`hdr`*`, INKMLoc` *`field`*`)`
Description	Retrieves a count of the number of values in the MIME field located at *`field`* within the marshal buffer *`bufp`*.
Returns	The number of values in the specified MIME field. `INK_ERROR` if error.
First release	Traffic Server 3.5

INKMimeHdrClone

Copies a MIME header and returns the location of the copy.

Prototype	`INKMLoc` **`INKMimeHdrClone`**`(INKMBuffer` *`dest_bufp`*`, INKMBuffer` *`src_bufp`*`, INKMLoc` *`src_hdr_loc`*`)`
Description	Copies the contents of the MIME header located at *`src_hdr_loc`* within the marshal buffer *`src_bufp`* to the marshal buffer *`dest_bufp`*.
Returns	The `INKMLoc` location of the copied header. Release the returned handle with a call to `INKHandleMLocRelease`. `INK_ERROR_PTR` if error.
First release	Traffic Server 3.5

INKMimeHdrCopy

Copies a MIME header to a specified MIME header location.

Prototype	`INKReturnCode` **`INKMimeHdrCopy`** `(INKMBuffer` *`dest_bufp`*`, INKMLoc` *`dest_hdr_loc`*`, INKMBuffer` *`src_bufp`*`, INKMLoc` *`src_hdr_loc`*`)`
Description	Copies the contents of the MIME header located at *`src_hdr_loc`* within the marshal buffer *`src_bufp`* to the MIME header located at *`dest_hdr_loc`* within the marshal buffer *`dest_bufp`*.`INKMimeHdrCopy` works correctly even if *`src_bufp`* and *`dest_bufp`* point to different marshal buffers. **Note**: Make sure that the destination marshal buffer *and* destination MIME header location have been created before copying. See the example below, illustrating copying a response MIME header.
Returns	`INK_SUCCESS` if successful. `INK_ERROR` if an error occurs.

Example

```
static void
copyResponseMimeHdr (INKCont pCont, INKHttpTxn pTxn)
{
    INKMBuffer respHdrBuf, tmpBuf;
    INKMLoc respHttpHdrLoc, tmpMimeHdrLoc;

    if ( !INKHttpTxnClientRespGet (pTxn, &respHdrBuf, &respHttpHdrLoc) )
{
        INKError ("couldn't retrieve client response header\n");
        INKHandleMLocRelease (respHdrBuf, INK_NULL_MLOC,
            respHttpHdrLoc);
        goto done;
    }
    tmpBuf = INKMBufferCreate ();
    tmpMimeHdrLoc = INKMimeHdrCreate(tmpBuf);

    INKMimeHdrCopy(tmpBuf, tmpMimeHdrLoc, respHdrBuf, respHttpHdrLoc);

    INKHandleMLocRelease (tmpBuf, INK_NULL_MLOC, tmpMimeHdrLoc);
    INKHandleMLocRelease (respHdrBuf, INK_NULL_MLOC, respHttpHdrLoc);

    INKMBufferDestroy(tmpBuf);

    done:
    INKHttpTxnReenable(pTxn, INK_EVENT_HTTP_CONTINUE); }
```

First release Traffic Server 3.0

INKMimeHdrCreate

Creates a MIME header.

Prototype `INKMLoc` **`INKMimeHdrCreate`** `(INKMBuffer` *`bufp`*`)`

Description Creates a new MIME header within the marshal buffer *`bufp`*.

Returns Location of the newly created MIME header. Release with a call to `INKHandleMLocRelease`. `INK_ERROR_PTR` if error.

First release Traffic Server 3.0

INKMimeHdrDestroy

Destroys a MIME header.

Prototype	`INKReturnCode` **`INKMimeHdrDestroy`** `(INKMBuffer` *`bufp`*`, INKMLoc` *`hdr_loc`*`)`
Description	Destroys the MIME header located at *`hdr_loc`* within the marshal buffer *`bufp`*. Release the `INKMLoc` handle *`hdr_loc`* with a call to `INKHandleMLocRelease`.
Returns	`INK_SUCCESS` if successful. `INK_ERROR` if an error occurs.
First release	Traffic Server 3.0

INKMimeHdrFieldFind

Finds fields in a MIME header.

Prototype	`INKMLoc` **`INKMimeHdrFieldFind`** `(INKMBuffer` *`bufp`*`, INKMLoc` *`loc`*`, const char*` *`name`*`, int` *`length`*`)`
Description	Retrieves a MIME field from within the MIME header located at *`loc`* within the marshal buffer *`bufp`*. The *`name`* and *`length`* parameters specify which field to retrieve. For each MIME field in the MIME header, a case insensitive string comparison is done between the field name and *`name`*. The *`length`* parameter specifies how long the string pointed to by *`name`* is. If *`length`* is `-1`, then *`name`* is assumed to be null-terminated. If the requested field cannot be found then `0` is returned.
Returns	The location of the retrieved MIME header. Release with a call to `INKHandleMLocRelease`. `INK_ERROR_PTR` if error.
First release	Traffic Server 3.0

INKMimeHdrFieldGet

Gets a field in a MIME header.

Prototype	`INKMLoc` **`INKMimeHdrFieldGet`** `(INKMBuffer` *`bufp`*`, INKMLoc` *`hdr_loc`*`, int` *`idx`*`)`
Description	Retrieves a MIME field from within the MIME header located at *`hdr_loc`* within the marshal buffer *`bufp`*. The *`idx`* parameter specifies which field to retrieve. The fields are numbered from `0` to `INKMimeHdrFieldsCount` (*`bufp`*, *`hdr_loc`*) `- 1`. If *`idx`* does not lie within that range then `0` will be returned.
Returns	The location of the MIME field from within the MIME header. Release with a call to `INKHandleMLocRelease`. `INK_ERROR_PTR` if error.
First release	Traffic Server 3.0

INKMimeHdrFieldRemove

Removes a field in a MIME header.

Prototype	`INKReturnCode` **`INKMimeHdrFieldRemove`** `(INKMBuffer` *`bufp`*`, INKMLoc` *`hdr_loc`*`, INKMLoc` *`field`*`)`
Description	Removes the MIME header located at *`field`* within the marshal buffer *`bufp`* from the MIME header located at *`hdr_loc`* within the marshal buffer *`bufp`*. If the specified field cannot be found in the list of fields associated with the header then nothing is done. After the call to `INKMimeHdrFieldDestroy`, you must release the `INKMLoc` handle *`field`* with a call to `INKHandleMLocRelease`. **Note**: removing the MIME field doesn't destroy the field, it only detaches it, hiding it from the printed output.The field can be reattached by calling `INKMimeHdrFieldAppend`.
Returns	`INK_SUCCESS` if successful. `INK_ERROR` if an error occurs.
First release	Traffic Server 3.0

INKMimeHdrFieldsClear

Clears all the fields of a MIME header.

Prototype	`INKReturnCode` **`INKMimeHdrFieldsClear`** `(INKMBuffer` *`bufp`*`, INKMLoc` *`hdr_loc`*`)`
Description	Removes and destroys all the MIME fields within the MIME header located at *`hdr_loc`* within the marshal buffer *`bufp`*. Make sure you release any corresponding `INKMLoc` or string handles using `INKHandleMLocRelease` or `INKHandleStringRelease`.
Returns	`INK_SUCCESS` if successful. `INK_ERROR` if an error occurs.
First release	Traffic Server 3.0

INKMimeHdrFieldsCount

Counts the fields in a MIME header.

Prototype	`int` **`INKMimeHdrFieldsCount`** `(INKMBuffer` *`bufp`*`, INKMLoc` *`hdr_loc`*`)`
Description	Obtains a count of the number of MIME fields within the MIME header located at *`hdr_loc`* within the marshal buffer *`bufp`*.
Returns	The number of fields within the specified MIME header. `INK_ERROR` if error.
First release	Traffic Server 3.0

INKMimeHdrLengthGet

Gets the length of a MIME header.

Prototype	`int` **`INKMimeHdrLengthGet`** `(INKMBuffer` *`bufp`*`, INKMLoc` *`hdr_loc`*`)`
Description	Calculates the length of the MIME header located at *`hdr_loc`* within the marshal buffer *`bufp`* if it were returned as a string. This is the length of the MIME header in its unparsed form.
Returns	The length of the specified MIME header. `INK_ERROR` if error.
First release	Traffic Server 3.0

INKMimeHdrParse

Parses a MIME header.

Prototype	`int` **`INKMimeHdrParse`** `(INKMimeParser` *`parser`*`,`
	`INKMBuffer` *`bufp`*`, INKMLoc` *`hdr_loc`*`,`
	`const char **`*`start`*`, const char *`*`end`*`)`

Description	Parses a MIME header. The MIME header must have already been allocated and both *`bufp`* and *`hdr_loc`* must point within that header. The *`start`* argument points to the current position of the buffer being parsed and the *`end`* argument points to **one byte after** the end of the buffer. On return, *`start`* is modified to point past the last character parsed. It is possible to parse a MIME header a single byte at a time using repeated calls to `INKMimeHdrParse`. As long as an error does not occur, the `INKMimeHdrParse` function will consume that single byte and ask for more.
Returns	`INK_PARSE_ERROR` is returned on error. `INK_PARSE_DONE` is returned when a `\r\n\r\n` pattern is encountered, indicating the end of the header. `INK_PARSE_CONT` is returned if parsing of the header stopped because the end of the buffer was reached.
First release	Traffic Server 3.0

INKMimeParserClear

Clears a MIME header parser so it may be reused.

Prototype	`INKReturnCode` **`INKMimeParserClear`** `(INKMimeParser` *`parser`*`)`
Description	Clears the specified MIME *`parser`* so it may be used again.
Returns	`INK_SUCCESS` if successful. `INK_ERROR` if an error occurs.
First release	Traffic Server 3.0

INKMimeParserCreate

Creates a parser for MIME headers.

Prototype	`INKMimeParser` **`INKMimeParserCreate`** `(void)`
Description	Creates a MIME parser. The parser's data structure contains information about the header being parsed. A single MIME parser can be used multiple times, though not simultaneously. Before being used again, the parser must be cleared by calling`INKMimeParserClear`.
Returns	A pointer to the newly created MIME parser. `INK_ERROR_PTR` if error.
First release	Traffic Server 3.0

INKMimeParserDestroy

Destroys a MIME header parser.

Prototype	`INKReturnCode` **`INKMimeParserDestroy`** `(INKMimeParser` *`parser`*`)`
Description	Destroys the specified MIME *parser* and frees the associated memory.
Returns	`INK_SUCCESS` if the parser is successfully destroyed. `INK_ERROR` if an error occurs.
First release	Traffic Server 3.0

INKMimeHdrPrint

Prints a MIME header to an IO buffer.

Prototype	`INKReturnCode` **`INKMimeHdrPrint`** `(INKMBuffer` *`bufp`*`, INKMLoc` *`hdr_loc`*`,` `INKIOBuffer` *`iobufp`*`)`
Description	Formats the MIME header located at *`hdr_loc`* within the marshal buffer *`bufp`* into the IO buffer *`iobufp`*. See *IO buffers, on page 128* for information on allocating an IO Buffer and retrieving data from within one.
Returns	`INK_SUCCESS` if successful. `INK_ERROR` if an error occurs.
First release	Traffic Server 3.0

Mutex functions

INKMutexCreate

Creates a new `INKMutex`.

Prototype	`INKMutex` **`INKMutexCreate`** `(void)`
Description	Creates a new `INKMutex`.

Returns	A handle to the newly created mutex. `INK_ERROR_PTR` if error.
First release	Traffic Server 3.0

INKMutexLock

Locks an `INKMutex`.

Prototype	`INKReturnCode` **`INKMutexLock`** `(INKMutex` *`mutexp`*`)`
Description	Locks the `INKMutex` *`mutexp`*. If *`mutexp`* is already locked then `INKMutexLock` will block until the mutex is unlocked. An `INKMutex` will be recursively locked if `INKMutexLock` is called on the same mutex twice from the same thread. That is, the following example will succeed and not block on the second call to `INKMutexLock`.
Returns	`INK_SUCCESS` if the mutex is successfully locked. `INK_ERROR` if an error occurs.

Example	`INKMutexLock (some_mutex);`
	`INKMutexLock (some_mutex);`
	`INKMutexUnlock (some_mutex);`
	`INKMutexUnlock (some_mutex);`

First release	Traffic Server 3.0

INKMutexLockTry

Tries to lock an `INKMutex`.

Prototype	`INKReturnCode` **`InkMutexLockTry`** `(INKMutex` *`mutex,`* `int` *`*lock`*`)`
Description	Tries to lock the `INKMutex` *`mutex`*. Information as to whether the lock was grabbed or not is set in `int` *`*lock`*. `INKReturnCode` will tell you if the call was successful or not, but does not indicate whether or not the lock was grabbed. In general, use `InkMutexLockTry` to obtain a mutex. See the example below.
Returns	If the mutex was successfully locked, `1` will be returned. If *`mutex`* is already locked then `0` will be returned.

Example

```
int handler (INKCont contp, INKEvent event, void *edata)
{
   //this continuation tries to grab a mutex
   int retval, lock = 0;
retvak = InkMutexLockTry (mutex, &lock);
   if (!lock)
   {
   /* Schedule a retry; RETRY_TIME should be 10 ms or longer. */
      INKContSchedule (contp, RETRY_TIME);
      return INK_EVENT_IMMEDIATE;
   }

   // Now the mutex is grabbed
   do_some_job ...
   INKMutexUnlock (mutexp);
}
```

First release Traffic Server 3.0

INKMutexUnlock

Unlocks an `INKMutex`.

Prototype `INKReturnCode` **`INKMutexUnlock`** `(INKMutex` *`mutexp`*`)`

Description Unlocks the `INKMutex` *`mutexp`*. If *`mutexp`* was recursively locked then `INKMutexUnlock` will not actually unlock the mutex but simply decrement the recursion count.

Returns `INK_SUCCESS` if the mutex is successfully unlocked.
`INK_ERROR` if an error occurs.

First release Traffic Server 3.0

Continuation functions

INKContCall

Calls a continuation.

Prototype `int` **`INKContCall`** `(INKCont` *`contp`*`, INKEvent` *`event`*`, void *`*`edata`*`)`

Description Sends *`event`* and *`edata`* to the *`contp`*'s handler function. It is an error to call a continuation without holding the continuation's lock.

Returns The values returned by the continuation *`contp`* event handler.

First release Traffic Server 3.0

INKContCreate

Creates a continuation.

Prototype	`INKCont` **`INKContCreate`** `(INKEventFunc` *`funcp`*`, INKMutex` *`mutexp`*`)`
Description	Creates a new `INKCont`. The continuation's handler function is *`funcp`*, and its mutex is *`mutexp`*. As mentioned previously, a continuation's mutex can be `NULL`. This is accomplished by specifying `NULL` for *`mutexp`*. **Note**: If you specify a NULL mutex, a mutex is created for the continuation and this mutex is held when the continuation is called back.
Returns	A handle to the newly created continuation. `INK_ERROR_PTR` if `INKCont` object is not successfully created.
First release	Traffic Server 3.0

INKContDataGet

Gets a data pointer from a continuation.

Prototype	`void*` **`INKContDataGet`** `(INKCont` *`contp`*`)`
Description	Retrieves the data pointer from *`contp`*. The data pointer can be set via a call to `INKContDataSet`. It is up to the plugin to allocate/deallocate the pointer.
Returns	The pointer on the continuation *`contp`* data, or `INK_ERROR_PTR` if error.
First release	Traffic Server 3.0

INKContDataSet

Sets a data pointer for a specified continuation.

Prototype	`INKReturnCode` **`INKContDataSet`** `(INKCont` *`contp`*`, void *`*`data`*`)`
Description	Sets the data pointer of *`contp`* to *`data`*. The data can later be retrieved by a call to `INKContDataGet`.
Returns	`INK_SUCCESS` if the pointer is successfully set. `INK_ERROR` if an error occurs.
First release	Traffic Server 3.0

INKContDestroy

Destroys a continuation.

Prototype	`INKReturnCode` **`INKContDestroy`** `(INKCont` *`contp`*`)`
Description	Destroys the continuation *`contp`*. `INKContDestroy` is used to destroy both continuations and vconnections (see *Vconnections, on page 121*). The internal continuation data structures are destroyed, but no attempt is made to guarantee that there are no outstanding references to this continuation.

Returns	`INK_SUCCESS` if the continuation is successfully destroyed. `INK_ERROR` if an error occurs.
First release	Traffic Server 3.0

INKContMutexGet

Gets the mutex for a specified continuation.

Prototype	`INKMutex` **`INKContMutexGet`** `(INKCont` *`contp`*`)`
Description	Gets the mutex for *`contp`*.
Returns	A handle to the mutex for the specified continuation. `INK_ERROR_PTR` if error.
First release	Traffic Server 3.0

INKContSchedule

Schedules a continuation to receive an event.

Prototype	`INKAction` **`INKContSchedule`** `(INKCont` *`contp`*`, unsigned int` *`timeout`*`)`
Description	Schedules the continuation represented by *`contp`* to receive an event. The *`timeout`* refers to a time in milliseconds from the present at which to send the event. When the *`contp`* is called back and if *`timeout`* is `0`, then the event sent will be `INK_EVENT_IMMEDIATE`. If *`timeout`* is greater than `0` then the event sent will be `INK_EVENT_TIMEOUT`.
Returns	An `INKAction` object. `INK_ERROR_PTR` if error.
First release	Traffic Server 3.0

Plugin configuration functions

INKConfigDataGet

Gets `configuration` data.

Prototype	`void*` **`INKConfigDataGet`** `(INKConfig` *`configp`*`)`
Description	Retrieves the data pointer from within the configuration pointer *`configp`*. Before you use `INKConfigDataGet`, you must give the configuration data an identifier with `INKConfigSet` and then retrieve the `INKConfig` pointer *`configp`* with a call to `INKConfigGet`. See the code snippet in the previous section.
First release	Traffic Server 3.0

INKConfigGet

Returns a pointer to the Traffic Edge configuration.

Prototype	`INKConfig` **`INKConfigGet`** `(unsigned int` *`id`*`)`
Description	Retrieves the current configuration pointer associated with the configuration identifier *`id`*. The function `INKConfigDataGet` can then be used to retrieve the data pointer from within the configuration. `INKConfigGet` increments the reference count inside the configuration. It is important to call `INKConfigRelease` to decrement the reference count when the user is done with the configuration pointer. Before you call `INKConfigGet`, you must set the identifier *`id`* to some plugin configuration data using `INKConfigSet`. See the code snippet in the previous section.
Returns	A pointer to the current Traffic Edge configuration.
First release	Traffic Server 3.0

INKConfigRelease

Releases a configuration pointer.

Prototype	`void` **`INKConfigRelease`** `(unsigned int` *`id`*`, INKConfig` *`configp`*`)`
Description	Releases the configuration pointer *`configp`* on the configuration associated with the identifier*`id`*. It is possible that*`configp`* is no longer the current configuration in which case `INKConfigRelease` may end up calling the configuration's destroy function. See the code snippet in the previous section.
First release	Traffic Server 3.0

INKConfigSet

Assigns an identifier to plugin configuration data.

Prototype	`unsigned int` **`INKConfigSet`** `(unsigned int` *`id`*`, void *`*`data`*`,` `INKConfigDestroyFunc` *`funcp`*`)`
Arguments	`unsigned int` *`id`* is the identifier that is assigned to configuration data. Do not use `1` or `2` for *`id`*. Traffic Edge internally assigns these IDs to parent and HTTP configurations. You can enter `0` as *`id`*, and `INKConfigSet` will allocate an identifier for you (with a value of 3 or greater). There is an internal upper limit of `100` on *`id`*. *`void *data`* points to the data that you are associating to *`id`*. `INKConfigDestroyFunc` *`funcp`* is a pointer to a destroy function that is called when Traffic Edge determines that there are no more references to *`data`*. The only argument of *`funcp`* is *`data`*.
Returns	The `unsigned int` that was assigned to the data. If the input *`id`* is `0` then a new configuration identifier is allocated (of value 3 or larger). If the input *`id`* is `0`, the return value is the available identifier allocated by Traffic Edge. If *`id`* is non-zero, the return value is *`id`*.

Description	Sets the opaque data pointer *data* to be associated with the configuration identifier *id*. If *id* is 0 then Traffic Edge allocates a new configuration identifier, and INKConfigSet returns this value. If *id* is non-zero, INKConfigSet returns *id*. To make sure that the configuration identifier stays within the recommended range of 3 to 100, follow the code example in the previous section. **Caution:** Never pick a configuration identifier yourself. When you need a new config *id*, you MUST always pass 0 as *id* to the INKConfigSet API which will return a new valid *id*. It is not safe to pick up a randomly selected id because there might be some conflict with *id*s already in use by Traffic Edge. This can cause severe memory corruption as the INKConfig mechanism is also used internally by Traffic Edge. The *funcp* parameter is a pointer to a destroy function which will be called with *data* as its only parameter when Traffic Edge determines that there are no more references to *data*. **Note**: *data* will not be destroyed while it is the current piece of configuration data since the current data always has a reference count of at least 1. See the code snippet in the previous section for usage.
First release	Traffic Server 3.0

Action functions

INKActionCancel

Cancels an action.

Prototype	INKReturnCode **INKActionCancel** (INKAction *actionp*)
Description	Cancels an INKAction. If a NULL argument is passed to INKActionCancel, Traffic Edge will crash and will not return INK_ERROR. Note that it is the programmer's responsibility to ensure that a non-null value is passed to INKActionCancel.
Returns	INK_SUCCESS if the action is successfully cancelled. INK_ERROR if an error occurs.
First release	Traffic Server 3.0

INKActionDone

Tells you if an action is completed.

Prototype	`int` **`INKActionDone`** `(INKAction` *`actionp`*`)`

Description	Is *`actionp`* a completed action. If a `NULL` argument is passed to `INKActionDone`, Traffic Edge will crash and will not return `INK_ERROR`. Note that it is the programmer's responsibility to ensure that a non-null value is passed to `INKActionDone`.
	Important: Always use `INKActionDone` immediately after the call that assigns the action. For example:
	`actionp = INKContSchedule(contp, SOME_TIMEOUT_VALUE);`
	`if (INKActionDone(actionp)){`
	`//event has already occurred`
	`}`
	If you call `INKActionDone(actionp)` some time later or some where else, it always returns false, and therefore does not accurately reflect whether the action is completed.

Returns	`0` if the action **has not** completed. `1` if the action has completed `INK_ERROR` if an error has occurred.
First release	Traffic Server 3.0

Host Lookup Functions

INKHostLookup

Asks Traffic Edge to do a DNS lookup of a host name.

Prototype	`INKAction INKHostLookupResult (INKCont` *`contp`*`, char *`*`hostname`*`,` `int` *`namelen`*`)`
Arguments	`INKCont` *`contp`* is the continuation that Traffic Edge calls back when the DNS lookup occurs. `char *`*`hostname`* is the name to look up. Null terminated. `int` *`namelen`* is the length of hostname +1 (add one to account for null termination).
Description	Initiates a DNS lookup of *`hostname`*. When the lookup occurs, Traffic Edge sends *`contp`* `INK_EVENT_DNS_LOOKUP`. If the lookup is successful (IP address resolved), the `void * data` passed to the handler of the continuation *`contp`* is a data of type `INKHostLookupResult`. You can then use `INKHostLookupResultIPGet` to convert this information to an unsigned int representing the IP address. If the lookup fails (IP address not resolved), the `void * data` passed to the handler of continuation *`contp`* is a null pointer. You have the option to cancel the action returned by `INKHostLookup` by using `INKActionCancel`. Note that reentrant calls are possible, i.e. the cache can call back the user (*`contp`*) in the same call.

Returns	An `INKAction` object if successful. `INK_ERROR_PTR` if an argument is incorrect or if the API fails.
First Release	Traffic Server 5.2

INKHostLookupResultIPGet

Gets the IP address of a host name that Traffic Edge has looked up.

Prototype	`InkReturnCode INKHostLookupResultIPGet (INKHostLookupResult` *`lookup_result`*`, unsigned int *`*`ip`*`)`
Arguments	`INKHostLookupResult` *`lookup_result`* is information returned by `INKHostLookupResult`. `unsigned int *`*`ip`* is set to the value of the IP address, in network byte order.
Description	Converts the information retrieved by `INKHostLookupResult` to an unsigned int representing the IP address.
Returns	`INK_SUCCESS` if the API is called successfully. `INK_ERROR` if an error occurs while calling the API or if an argument is invalid.
First Release	Traffic Server 5.2

Vconnection functions

INKVConnAbort

Closes a vconnection and specifies that the operations it was performing were aborted.

Prototype	`INKReturnCode` **`INKVConnAbort`** `(INKVConn` *`connp`*`, int` *`error`*`)`
Description	Closes the vconnection *`connp`* and specifies that the operations it was performing were aborted. The vconnection will be de-allocated at some point in the near future after having `INKVConnAbort` called upon it. After calling `INKVConnClose`, a user will not receive any more events from *`connp`*. For most vconnections, `INKVConnClose` and `INKVConnAbort` perform identical operations. A potential difference is that when a vconnection is aborted the vconnection implementor can decide to do something special. For instance, a vconnection writing a file to disk might decide to delete the file.
Returns	`INK_SUCCESS` if the connection is successfully aborted. `INK_ERROR` if an error occurs.
First release	Traffic Server 3.0

INKVConnClose

Closes a vconnection.

Prototype	`INKReturnCode` **`INKVConnClose`** `(INKVConn` *`connp`*`)`
Description	Closes the vconnection *`connp`*. The vconnection will be de-allocated at some point in the near future after having `INKVConnClose` called upon it. After calling `INKVConnClose`, a user will not receive any more events from *`connp`*.

Returns	`INK_SUCCESS` if the connection is successfully closed. `INK_ERROR` if an error occurs.
First release	Traffic Server 3.0

INKVConnClosedGet

Gets a closed vconnection.

Prototype	`INKReturnCode` **`INKVConnClosedGet`** `(INKVConn` *`connp`*`)`
Description	Retrieves the closed status for a vconnection. `INKVConnClosedGet` is intended to be used by vconnection implementors and not by vconnection users. It is not safe for a vconnection user to call `INKVConnClosedGet` since if the vconnection actually is closed then it is possible (and likely) for it to be de-allocated at any time. **Note:** This API can be used **ONLY** on transformation VConnections. **NEVER** use it on Cache VConnections, Net VConnections or any other type of VConnections.
Returns	`INK_SUCCESS` if successful. `INK_ERROR` if an error occurs.
First release	Traffic Server 3.0

INKVConnRead

Reads a vconnection.

Prototype	`INKVIO` **`INKVConnRead`** `(INKVConn` *`connp`*`, INKCont` *`contp`*`, INKIOBuffer` *`bufp`*`, int` *`nbytes`*`)`
Description	Initiates a read operation on the vconnection *`connp`*. The read operation writes into the buffer *`bufp`*. The continuation *`contp`* will be called back with either `INK_EVENT_ERROR`, `INK_EVENT_VCONN_READ_READY`, `INK_EVENT_VCONN_READ_COMPLETE` or `INK_EVENT_VCONN_EOS`. Refer to *The vconnection user's view, on page 121* for more information about these events. The number of bytes to read is specified by the *`nbytes`* parameter.
Returns	A handle to the vconnection. `INK_ERROR_PTR` if an error occurs.
First release	Traffic Server 3.0

INKVConnReadVIOGet

Obtains the output VIO for a vconnection.

Prototype	`INKVIO` **`INKVConnReadVIOGet`** `(INKVConn` *`connp`*`)`
Description	Retrieves the read VIO for a vconnection. `INKVConnReadVIOGet` is intended to be used by vconnection implementors and not by vconnection users. Note that this API can only be used for transformations. It is not used for NetVConn or CacheVConn.

Returns	A handle to the vconnection. `INK_ERROR_PTR` if an error occurs.
First release	Traffic Server 3.0

INKVConnShutdown

Shuts down a vconnection.

Prototype	`INKReturnCode` **`INKVConnShutdown`** `(INKVConn` *`connp`*`, int` *`read`*`, int` *`write`*`)`
Description	Shuts down a portion of the vconnection *`connp`*. If *`read`* is non-zero, then the read portion of *`connp`* is shutdown indicating that the user does not want to be called back regarding any more read events on this vconnection. If *`write`* is non-zero, then the write portion of *`connp`* is shutdown indicating that the user does not want to be called back regarding any more write events on this vconnection.
Returns	`INK_SUCCESS` if the connection is successfully shutdown. `INK_ERROR` if an error occurs.
First release	Traffic Server 3.0

INKVConnWrite

Writes a vconnection.

Prototype	`INKVIO` **`INKVConnWrite`** `(INKVConn` *`connp`*`, INKCont` *`contp`*`, INKIOBufferReader` *`readerp`*`, int` *`nbytes`*`)`
Description	Initiates a write operation on the vconnection *`connp`*. The write operation reads from the buffer reader *`readerp`*. The continuation *`contp`* will be called back with either `INK_EVENT_ERROR`, `INK_EVENT_VCONN_WRITE_READY`, or `INK_EVENT_VCONN_WRITE_COMPLETE`. Refer to *The vconnection user's view, on page 121* for more information about these events. The number of bytes to write is specified by the *`nbytes`* parameter.
Returns	A handle to the vconnection. `INK_ERROR_PTR` if an error occurs.
First release	Traffic Server 3.0

INKVConnWriteVIOGet

Obtains the input VIO for a vconnection.

Prototype	`INKVIO` **`INKVConnWriteVIOGet`** `(INKVConn` *`connp`*`)`
Description	Retrieves the write VIO for a vconnection. `INKVConnWriteVIOGet` is intended to be used by vconnection implementors and not by vconnection users. Note that this API can only be used for transformations.
Returns	A handle to the vconnection. `INK_ERROR_PTR` if error.
First release	Traffic Server 3.0

Netvconnection functions

INKNetAccept

Accepts a TCP/IP connection on a specified port.

Prototype	`INKAction` **`INKNetAccept`** `(INKCont` *`contp`*`, int` *`port`*`)`
Arguments	`INKCont` *`contp`* is the continuation that is called back when a connection is accepted. `int` *`port`* is the port to listen to for incoming TCP/IP connections.
Description	Accepts a TCP/IP connection on *`port`*. When Traffic Edge receives a connection on a specified port, it calls back *`contp`* with the event `INK_EVENT_NET_ACCEPT or INK_EVENT_NET_ACCEPT_FAILED` If event is `INK_EVENT_NET_ACCEPT`, the `void *` *`data`* passed to the handler of the continuation *`contp`* is a data of type NetVConnection representing the connection. If event is `INK_EVENT_NET_ACCEPT_FAILED`, it means an attempt of connection was aborted or failed. The plugin should just return from the continuation's handler. The user (*`contp`*) has the option to cancel the action returned by `INKNetAccept` by using `INKActionCancel`.
Returns	An `INKAction` object if successful. `INK_ERROR_PTR` if an argument was incorrect or if the API failed.
First Release	Traffic Server 5.2

INKNetConnect

Initiate a network connection to a server.

Prototype	`INKAction` **`INKNetConnect`** `(INKCont` *`contp`*`, unsigned int` *`ip`*`, int` *`port`*`)`
Arguments	`INKCont` *`contp`* is the continuation to be associated with the connection. `int` *`ip`* is the IP address , in network byte order, of the host to connect to. `int` *`port`* is port number for the host, specified in network byte order.
Description	Opens up a network connection to the host specified by *`ip`* on the port specified by *`port`*. T If the connection is successfully opened, *`contp`* will be called back with the event `INK_EVENT_NET_CONNECT` and the new network vconnection will be passed in the event data parameter. If the connection is not successful, *`contp`* will be called back with the event `INK_EVENT_NET_CONNECT_FAILED`. **Note**: It's possible to receive `INK_EVENT_NET_CONNECT` even if the connection failed, because of the implementation of network sockets in the underlying operating system. There is an exception: if a plugin tries to open a connection to a port on its own host machine, then `INK_EVENT_NET_CONNECT` is sent only if the connection is successful. In general, however, your plugin needs to look for `INK_EVENT_VCONN_WRITE_READY` or `INK_EVENT_VCONN_READ_READY` to make sure that the connection is successfully opened. Note that reentrant calls are possible, i.e. the net processor can call back the user (*`contp`*) in the same call.
Returns	An `INKAction` object.
First release	Traffic Server 3.0

INKNetVConnRemoteIPGet

Retrieves the remote host's IP address.

Prototype	`INKReturnCode INKNetVConnRemoteIPGet (INKVConn vc, unsigned int *ip)`
Arguments	`INKVConn vc` is the connection between Traffic Edge and the other end of the connection (can be remote client or server). `unsigned int *ip` is set to the remote IP address in network byte order.
Description	Obtains the remote IP address in network byte order.
Returns	`INK_SUCCESS` if API is called successfully. `INK_ERROR` if an error occurs while calling the API or if an argument is invalid. Note this returns IP in IP Version 4.
First release	Traffic Server 5.2

INKNetVConnRemotePortGet

Retrieves the remote host's port number.

Prototype	`InkReturnCode INKNetVConnRemotePortGet (INKVConn vc, int *port)`
Arguments	`INKVConn vc` is the connection between Traffic Edge and the other end of the connection (can be remote client or server). `int *port` is set to the remote port value in host byte order.
Description	Obtains the port number of the remote host for the specified connection. The port is returned in host byte order.
Returns	`INK_SUCCESS` if API is called successfully. `INK_ERROR` if an error occurs while calling the API or if an argument is invalid.
First release	Traffic Server 5.2

Cache interface functions

INKCacheKeyCreate

Creates a new cache key to be assigned to an object to be cached.

Prototype	`INKReturnCode` **`INKCacheKeyCreate`**`(InkCacheKey *new_key)`
Arguments	`INKCacheKey *new_key` is set to the allocated key.
Description	Creates (allocates memory for) a new cache key. The key can then be generated and assigned to an object using `INKCacheKeyDigestSet`.
Returns	`INK_SUCCESS` if success. `INK_ERROR` if cache key could not be allocated.
First Release	Traffic Server 5.2

INKCacheKeyDigestSet

Generates and assigns a cache key to an object to be cached.

Prototype `INKReturnCode` **`INKCacheKeyDigestSet`**`(INKCacheKey` *`key`*`,`
`const unsigned char *`*`input`*`, int` *`length`*`)`

Arguments `INKCacheKey` *`key`* is the key to be associated to the cached object. Before calling `INKCacheKeyDigestSet` you must create the key with `INKCacheKeyCreate`. Note that in order to generate unique keys, you must use unique input strings. In other words, if the input strings are identical, INKCacheKeyCreate will generate identical keys.

`const unsigned char *`*`input`* is a character string that uniquely identifies the object. In most cases, it is the URL of the object.

`int` *`length`* is the length of the string *`input`*.

Description Generates and assigns a cache key to an object to be cached.

Returns `INK_SUCCESS` if the cache key was successfully generated.

`INK_ERROR` if digest could not be set.

Example
```
const char *digest_string = "mydigest"
INKCacheKey mykey;
INKCacheKeyCreate(&mykey);
INKCacheKeyDigestSet(mykey,digest_string, strlen(digest_string);
```

First Release Traffic Server 5.2

INKCacheKeyHostNameSet

Associates a host name to a cache key. Use if you want to support cache partitioning by host name.

Prototype `INKReturnCode` **`INKCacheKeyHostNameSet`**`(INKCacheKey` *`key`*`,`
`const unsigned char *`*`hostname`*`, int` *`host_len`*`;`

Arguments `INKCacheKey` *`key`* is the key to the cached object.

`const unsigned char *`*`hostname`* is the host name you are associating to the cache key.

`int` *`host_len`* is the length of the string *`hostname`*.

Description Associates a host name to a cache key. The host name setting is used in conjunction with the TS config file `partition.config` and `hosting.config` that allows you to specify under which cache partition the object should be stored.

Returns `INK_SUCCESS` if the host name was successfully associated with the cache key.

`INK_ERROR` if hostname could not be set or is invalid.

First Release Traffic Server 5.2

INKCacheKeyDestroy

Destroys a cache key.

Prototype `INKReturnCode` **`INKCacheKeyDestroy`**`(INKCacheKey` *`key`*`)`

Arguments `INKCacheKey` *`key`* is the key to be destroyed.

Description Destroys a cache key (deallocate memory). You must destroy cache keys when you are finished with them (after all reads and writes are completed).

Returns `INK_SUCCESS` if the cache key was successfully destroyed.

`INK_ERROR` if key could not be deallocated or was not valid.

First Release Traffic Server 5.2

INKCacheRead

Initiates a cache read or lookup of an object in the Traffic Edge cache.

Prototype `INKAction` **`INKCacheRead`** `(INKCont` *`contp`*`, INKCacheKey` *`key`*`)`

Arguments `INKCont` *`contp`* is the continuation that the cache calls back (telling it either the object exists and can be read or not).

`INKCacheKey` *`key`* is the cache key corresponding to the object to be read.

Description Asks the Traffic Edge cache if the object corresponding to *`key`* exists in the cache and can be read.

You can do a cache lookup to determine whether or not an object is in the cache. To do a cache lookup, call `INKCacheRead` on a continuation *`contp`*. If the object can be read, the cache calls *`contp`* back with the event `INK_EVENT_CACHE_OPEN_READ`. In this case, the cache also passes *`contp`* a cache vconnection and *`contp`* can then initiate a read operation on that vconnection using `INKVConnRead`. `INKVConnCacheObjectSizeGet` can be used to determine the size of the object in the cache.

If the object cannot be read (if, for instance, it is not in the cache), the cache calls *`contp`* back with the event `INK_EVENT_CACHE_OPEN_READ_FAILED`. An error code is passed in the void *edata argument of *`contp`*. The error code can be:

INK_CACHE_ERROR_NOT_READY: Trying to access to the cache while it's not yet initialized.

INK_CACHE_ERROR_NO_DOC: Document does not exist in cache.

INK_CACHE_ERROR_DOC_BUSY: Trying to read a document while another continuation is writing on it.

Any other value: unknown read failure

Finally, once you have performed a cache lookup, you can write into cache with INKCacheWrite. The user (*`contp`*) also has the option to cancel the action returned by `INKCacheRead` by using `INKActionCancel`.

Note: It is up to the user to read the data from the `cache vc iobuffer` and consume it. The cache does not bufferize the data. The cache will not call the user back unless all the data from the `cache iobuffer` is consumed.

Note that reentrant calls are possible; in other words, the cache can call back the user (*`contp`*) in the same call.

Returns An `INKAction` object if successful.

`INK_ERROR_PTR` if an argument is incorrect or if the API failed.

First Release Traffic Server 5.2

INKCacheReady

Determines if the Traffic Edge cache is initialized and ready to accept requests for the specified data type.

Prototype	`INKReturnCode` **`INKCacheReady`** `(int *is_ready)`
Arguments	*`int *is_ready`* is the argument set to non-zero if cache ready and 0 if cache not ready.
Description	Asks the Traffic Edge cache if it is initialized and ready to accept requests. If the cache is not initialized, any attempt to read, write or remove document will fail. When a plugin starts (its *`INKPluginInit`* function is called), there is no guarantee that the cache is already initialized. This API is useful if a plugin needs to access to the cache from the *`INKPluginInit`* function. If the cache is not ready, the plugin should retry later.
Returns	`INK_SUCCESS` if API is called successfully. `INK_ERROR` if cache ready could not be set or is invalid.
First Release	Traffic Server 5.2

INKCacheWrite

Initiates writing an object to the Traffic Edge cache.

Prototype	`INKAction` **`INKCacheWrite`** `(INKCont` *`contp`*`, INKCacheKey` *`key`*`)`
Arguments	`INKCont` *`contp`* is the continuation that the cache calls back (telling it whether the write operation can proceed or not). `INKCacheKey` *`key`* is the cache key corresponding to the object to be cached.
Description	Asks the Traffic Edge cache if *`contp`* can start writing the object (corresponding to *`key`*) to the cache. If the object can be written, the cache calls *`contp`* back with the event `INK_EVENT_CACHE_OPEN_WRITE`. In this case, the cache also passes *`contp`* a cache vconnection in the `void *edata` argument and *`contp`* can then initiate a write operation on that vconnection using `INKVConnWrite`. The object is not committed to the cache until the vconnection is closed. If the object cannot be written, the cache calls *`contp`* back with the event `INK_EVENT_CACHE_OPEN_WRITE_FAILED`. This can happen, for example, if there is another object with the same key being written to the cache. An error code is passed in the void *edata argument of contp. The error code can be: `INK_CACHE_ERROR_NOT_READY`: Trying to access to the cache while it's not yet initialized. `INK_CACHE_ERROR_DOC_BUSY`: Trying to write a document while another continuation is writing or reading it. Any other value: unknown write failure. The user (*`contp`*) has the option to cancel the action returned by `INKCacheWrite`. The actual data is written/read to the cache through the cache vconnection. When the cache calls the user back with `OPEN_READ` or `OPEN_WRITE`, it passes a `INKVConn` to the user. The user uses this vconnection for any data transfer. When all data has been transferred, the user must do a `INKVConnClose`. In case of any errors, the user *must* do an `INKVConnAbort(`*`contp`*`, 0)`. Note: reentrant calls are possible; in other words, the cache can call back the user (*`contp`*) in the same call. Note: `INKCacheWrite` does not overwrite content already stored in the cache under the same cache key. If you try to do so, the cache returns `INK_EVENT_CACHE_OPEN_WRITE_FAILED`. To overwrite content, first call `INKCacheRemove` to remove the content, then call `INKCacheWrite`.

Returns An `INKAction` object if successful.

`INK_ERROR_PTR` if an argument is incorrect or the API fails.

First Release Traffic Server 5.2

INKCacheRemove

Removes an object from the Traffic Edge cache.

Prototype `INKAction` **`INKCacheRemove`** `(INKCont` *`contp`*`, INKCacheKey` *`key`*`)`

Arguments `INKCont` *`contp`* is the continuation that the cache calls back reporting the success or failure of the remove.

`INKCacheKey` *`key`* is the cache key corresponding to the object to be removed.

Description Removes the object corresponding to *`key`* from the cache.

If the object was removed successfully, the cache calls *`contp`* back with the event `INK_EVENT_CACHE_REMOVE`.

If the object was not found in the cache, the cache calls *`contp`* back with the event `INK_EVENT_CACHE_REMOVE_FAILED`. An error code is passed in the `void *edata` argument of *`contp`*. The error code can be:

INK_CACHE_ERROR_NOT_READY: Trying to access to the cache while it's not yet initialized.

INK_CACHE_ERROR_NO_DOC: Doc doesn't exist in cache

any other value: unknown remove failure

In both of these callbacks, the user does not have to do anything. The user does not get any vconnection from the cache, since no data needs to be transferred. When the cache calls the user back with `INK_EVENT_CACHE_REMOVE`, the remove has already been committed.

Note that reentrant calls are possible, i.e. the cache can call back the user (*`contp`*) in the same call.

Returns An `INKAction` object if successful.

`INK_ERROR_PTR` if an argument is incorrect or if the API fails.

First Release Traffic Server 5.2

INKCacheKeyPinnedSet

Pins the document corresponding to the specified key in the cache so that the garbage collection process will not delete the document from the cache for the specified number of seconds.

Prototype `INKReturnCode INKCacheKeyPinnedSet (INKCacheKey` *`key`*`, time_t` *`pin_in_cache`*`)`

Arguments `INKCacheKey` *`key`* is the cache key for the document to be pinned.

`time_t` *`pin_in_cache`* represents the number of seconds the document is to be pinned in the cache.

Description	Pins the document corresponding to the specified key in the cache for the specified number of seconds specified in *pin_in_cache*. Once the document is pinned, the garbage collection will not delete this document from the specifed number of seconds and the document can even persist across Traffic Edge re-runs. However, after the *pin_in_cache* interval has expired, the cache may delete the document at any time in order to reclaim space. To delete this document before the *pin_in_cache* interval expires, call the INKCacheRemove() function with the document's cache key. *InkCacheKeyPinnedSet()* should be used after a key is created and before writing the document to cache using *INKCacheWrite()*. By default, a document is not pinned in the cache and so can be garbage collected at anytime. Note that it is important that the records.config variable *proxy.config.cache.permit.pinning* be set to 1 in records.config to enable pinning.
Returns	INK_SUCCESS if the specified object was successfully pinned in the cache. INK_ERROR if the pin could not be set or is invalid.
First Release	Traffic Server 5.2

INKVConnCacheObjectSizeGet

Gets the size of the object in the cache.

Prototype	INKReturnCode **INKVConnCacheObjectSizeGet** (INKVConn *connp*, int **obj_size*)
Arguments	INKConn *connp* is the vconnection to the cache. int **obj_size* is set to the object size.
Description	When a cached object is requested from the cache (using INKCacheRead), and if the cache open was successful, this function can be called to get the size of the object in the cache.
Returns	INK_SUCCESS if API is called successfully. INK_ERROR if an error occurs while calling the API or if an argument is invalid.
First Release	Traffic Edge 4.0

Transformation functions

INKTransformCreate

Creates a transformation vconnection.

Prototype	INKVConn **INKTransformCreate** (INKEventFunc *event_funcp*, INKHttpTxn *txnp*)
Description	Creates a new transformation INKVConn. The vconnection's handler function is *funcp* and its mutex is taken from *txnp*.
Returns	The newly created transformation connection.
Example	See *The sample null transform plugin, on page 43*.
First release	Traffic Server 3.0

INKTransformOutputVConnGet

Retrieves the downstream (output) vconnection for a transformation.

Prototype	`INKVConn` **`INKTransformOutputVConnGet`** `(INKVConn` *`connp`*`)`
Description	Retrieves the output vconnection for the transformation *`connp`*. The output vconnection may be `NULL` if `INKTransformOutputVConnGet` is called before the write operation is initiated on *`connp`*. This is normally not an issue since a transformation would not want to output data until it has data input into it.
Returns	The downstream vconnection for the transformation. `INK_ERROR_PTR` if error.
First release	Traffic Server 3.0

VIO functions

INKVIOBufferGet

Gets a VIO buffer.

Prototype	`INKIOBuffer` **`INKVIOBufferGet`** `(INKVIO` *`viop`*`)`
Description	Gets the buffer for the IO operation described by *`viop`*. `INKVIOBufferGet` is used by vconnections performing read operations. Read operations write into their buffers.
Returns	The buffer for the specified IO operation. `INK_ERROR_PTR` if an error occurs.
First release	Traffic Server 3.0

IINKVIOVConnGet

Gets a VIO connection.

Prototype	`INKVConn` **`INKVIOVConnGet`** `(INKVIO` *`viop`*`)`
Description	Gets the vconnection associated with the IO operation described by *`viop`*. This is the vconnection passed to `INKVConnRead` or `INKVConnWrite`.
Returns	The vconnection for the specified IO operation. `INK_ERROR_PTR` if an error occurs.
First release	Traffic Server 3.0

INKVIOContGet

Gets an `INKVIOCont`.

Prototype	`INKCont` **`INKVIOContGet`** `(INKVIO` *`viop`*`)`
Description	Gets the continuation (user) for the IO operation described by *`viop`*. This is the continuation that the vconnection will call back when progress is made on the IO operation.
Returns	The continuation for the specified IO operation. `INK_ERROR_PTR` if an error occurs.
First release	Traffic Server 3.0

INKVIOMutexGet

Returns the mutex for the specified IO operation.(

Prototype	`INKMutex` **`INKVIOMutexGet`** `(INKVIO` *`viop`*`)`
Description	Gets the mutex for the IO operation described by *`viop`*. The mutex for the IO operation protects the buffer and continuation and other VIO members from simultaneous access. The vconnection implementor must obtain the mutex for a VIO before accessing any of its members. Since the VIO mutex is the same as the continuation's mutex, the vconnection user already holds the mutex whenever he is running and does not have to worry about grabbing it. For information on why vconnection transformations do not have to worry about grabbing the VIO mutex before accessing their write VIO, see *Transformations, on page 124*.
Returns	The mutex for the specified IO operation. `INK_ERROR_PTR` if an error occurs.
First release	Traffic Server 3.0

INKVIONBytesGet

Returns the number of bytes associated with a specified IO operation.

Prototype	`int` **`INKVIONBytesGet`** `(INKVIO` *`viop`*`)`
Description	Gets the number of bytes to be performed by the IO operation described by *`viop`*. This is the *`nbytes`* parameter passed to `INKVConnRead` or `INKVConnWrite`.
Returns	The number of bytes associated with the specified IO operation. `INK_ERROR` if an error occurs.
First release	Traffic Server 3.0

INKVIONBytesSet

Sets the number of bytes for the specified IO operation.

Prototype	`INKReturnCode` **`INKVIONBytesSet`** `(INKVIO` *`viop`*`, int` *`nbytes`*`)`
Description	Sets the number of bytes to be performed by the IO operation described by *`viop`*. Only the user of a vconnection should call `INKVIONBytesSet` and then, only carefully. `INKVIONBytesSet` should only be used to set the number of bytes to be done by the IO operation to a value that is greater than or equal to `INKVIONDoneGet`. The common usage of this function is to indicate to a vconnection that enough IO has been performed. By setting *`nbytes`* to the number done and re-enabling the operation, the user can indicate to the vconnection that the operation has completed.
Returns	`INK_SUCCESS` if the number of bytes associated with the IO operation is successfully set. `INK_ERROR` if an error occurs.
First release	Traffic Server 3.0

INKVIONDoneGet

Returns the number of bytes completed for the specified IO operation.

Prototype	`int` **`INKVIONDoneGet`** `(INKVIO` *`viop`*`)`
Description	Gets the number of bytes that have been completed on the IO operation described by *`viop`*. The number of completed bytes is also the number of bytes consumed out of or produced into the buffer passed to the IO operation.
Returns	The number of bytes that have been completed in the specified IO operation. `INK_ERROR` if an error occurs.
First release	Traffic Server 3.0

INKVIONDoneSet

Sets the number of bytes completed for the specified IO operation.

Prototype	`INKReturnCode` **`INKVIONDoneSet`** `(INKVIO` *`viop`*`, int` *`ndone`*`)`
Description	Sets the number of bytes that have been completed on the IO operation described by *`viop`* to *`ndone`*. Only vconnection implementors should call `INKVIONDoneSet`.
Returns	`INK_SUCCESS` if the number of completed bytes associated with the IO operation is successfully set. `INK_ERROR` if an error occurs.
First release	Traffic Server 3.0

INKVIONTodoGet

Returns the number of bytes remaining for the specified IO operation.

Prototype	`int` **`INKVIONTodoGet`** `(INKVIO` *`viop`*`)`
Description	Gets the number of bytes left to do on the IO operation described by *`viop`*. The number of bytes left to do is equal to the total number of bytes to perform on the IO operation minus the number that have been done. `INKVIONTodoGet` is a convenience function.
Returns	The number of bytes left that are associated with the specified IO operation. `INK_ERROR` if an error occurs.
Example	`INKVIONTodoGet (viop) == INKVIONBytesGet (viop) - INKVIONDoneGet (viop);`
First release	Traffic Server 3.0

INKVIOReaderGet

Obtains the buffer reader for the specified IO operation.

Prototype	`INKIOBufferReader` **`INKVIOReaderGet`** `(INKVIO` *`viop`*`)`
Description	Gets a buffer reader for the IO operation described by *`viop`*. `INKVIOReaderGet` is used by vconnections performing write operations. Write operations read from their buffers.
Returns	The buffer reader for the specified IO operation. `INK_ERROR_PTR` if an error occurs.
First release	Traffic Server 3.0

INKVIOReenable

Re-enables a VIO.

Prototype	`INKReturnCode` **`INKVIOReenable`** `(INKVIO` *`viop`*`)`
Description	Re-enables the vconnection associated with *`viop`*. Re-enabling the vconnection means that the vconnection will wake up and be able to determine that the buffer being used in its IO operation has changed.
Returns	`INK_SUCCESS` if the vconnection successfully re-enables. `INK_ERROR` if an error occurs.
First release	Traffic Server 3.0

IO buffer interface

INKIOBufferBlockNext

Gets next IO buffer block.

Prototype	`INKIOBufferBlock` **`INKIOBufferBlockNext`** `(INKIOBufferBlock` *`blockp`*`)`
Description	Gets the next block in the buffer block chain.
Returns	The next IO buffer block. `INK_ERROR_PTR` if an error occurs.
First release	Traffic Server 3.0

INKIOBufferBlockReadAvail

Indicates the number of IO buffer bytes available for reading.

Prototype	`int` **`INKIOBufferBlockReadAvail`** `(INKIOBufferBlock` *`blockp,`* `INKIOBufferReader` *`readerp`*`)`
Description	Obtains the number of bytes available for reading in the IO buffer block *`blockp`*. The *`readerp`* parameter is needed since each IO buffer reader maintains its own current offset.
Returns	The number of bytes available for reading in the IO buffer block. `INK_ERROR` if an error occurs.
First release	Traffic Server 3.0

INKIOBufferBlockReadStart

Starts reading IO buffer block.

Prototype	`const char*` **`INKIOBufferBlockReadStart`** `(INKIOBufferBlock` *`blockp`*`,` `INKIOBufferReader` *`readerp`*`, int *`*`avail`*`)`
Description	Gets the start point for reading from the IO buffer block *`blockp`*. The *`readerp`* parameter is needed since each IO buffer reader maintains its own current offset. `INKIOBufferBlockReadStart` stores the amount of data available for reading in the parameter *`avail`*. This is the same value that `INKIOBufferBlockReadAvail` returns. If *`avail`* is `NULL` then no attempt is made to de-reference it. **Note**: The *`avail`* parameter stores the amount of data available for reading on the specified `INKIOBufferBlock`. If you need to read all available data in an `INKIOBuffer`, make sure that your code keeps checking `INKIOBufferBlocks` until all the available data is read.
Returns	A pointer to the starting point for reading from the specified IO buffer block. `INK_ERROR_PTR` in case of an error.

Example Here is a sample routine, `transform_read_status_event` (modified from `server-transform.c`). It attempts to read a certain number of bytes. It calls `INKIOBufferBlockReadStart` to determine the number of bytes available to read (and get the start point within the `INKIOBufferBlock` to start reading). However, `INKIOBufferBlockReadStart` returns the available bytes within the current block only. The `INKIOBuffer` data structure contains a linked list of `INKIOBufferBlocks`, and so the available data within the `INKIOBuffer` could span more than one `INKIOBufferBlock`. The correct way to code this subroutine is to keep checking `INKIOBufferBlocks` for available data until all of the available `INKIOBuffer` data is read.

```
static int
transform_read_status_event (INKCont contp, TransformData *data,
                             INKEvent event, void *edata)
{
    switch (event) {
    case INK_EVENT_ERROR:
    case INK_EVENT_VCONN_EOS:
        return transform_bypass (contp, data);
    case INK_EVENT_VCONN_READ_COMPLETE:
        if (INKIOBufferReaderAvail (data->output_reader) ==
            sizeof (int)) {
            INKIOBufferBlock blk;
       char *buf;
            void *buf_ptr;
            int avail;
       int read_nbytes = sizeof (int);
       int read_ndone = 0;

        buf_ptr = &data->content_length;
        while (read_nbytes > 0) {
       blk = INKIOBufferReaderStart (data->output_reader);
       buf = (char *)INKIOBufferBlockReadStart (blk,
                    data->output_reader,
                    &avail);
       read_ndone = (avail >= read_nbytes)? read_nbytes : avail;
       memcpy (buf_ptr, buf, read_ndone);
       if (read_ndone > 0) {
           INKIOBufferReaderConsume (data->output_reader,
                       read_ndone);
           read_nbytes -= read_ndone;
           /* move ptr frwd by read_ndone bytes */
           buf_ptr = (char*)buf_ptr + read_ndone;
       }
        }
        data->content_length = ntohl (data->content_length);
        return transform_read (contp, data);
        }
        return transform_bypass (contp, data);
    default:
        break;
```

First release	Traffic Server 3.0

INKIOBufferBlockWriteAvail

Indicates the number of IO buffer bytes available for writing.

Prototype	`int` **`INKIOBufferBlockWriteAvail`** `(INKIOBufferBlock` *`blockp`*`)`
Description	Returns the number of bytes available for writing in the IO buffer block *`blockp`*.
Returns	The number of bytes available for writing. `INK_ERROR` if an error occurs.
First release	Traffic Server 3.0

INKIOBufferBlockWriteStart

Starts to write IO buffer block.

Prototype	`char*` **`INKIOBufferBlockWriteStart`** `(INKIOBufferBlock` *`blockp`*`, int *`*`avail`*`)`
Description	Gets the start point for writing into the IO buffer block *`blockp`*. The amount of data available for writing is stored in the parameter *`avail`*. This is the same value as would be returned by. If *`avail`* is `NULL` then no attempt is made to de-reference it.
Returns	A pointer to the starting point for writing to the specified IO buffer block. `INK_ERROR_PTR` if an error occurs.
First release	Traffic Server 3.0

INKIOBufferCopy

Copies an IO buffer.

Prototype	`int` **`INKIOBufferCopy`** `(INKIOBuffer` *`bufp`*`, INKIOBufferReader` *`readerp`*`, int` *`length`*`, int` *`offset`*`)`
Description	Copies *`length`* bytes of data from the IO buffer reader *`readerp`* to the IO buffer *`bufp`*. As described above, `INKIOBufferCopy` does not actually copy the data but simply copies pointers and adjusts reference counts appropriately. The parameter *`offset`* specifies the offset from *`readerp`*'s current position to start copying from.
Returns	The number of bytes actually copied. `INK_ERROR` if an error occurs.
First release	Traffic Server 3.0

INKIOBufferCreate

Creates an IO buffer.

Prototype	`INKIOBuffer` **`INKIOBufferCreate`** `(void)`
Description	Creates a new IO Buffer. The IO buffer is initially empty.

Returns	A handle to the newly created IO buffer. `INK_ERROR_PTR` if an error occurs.
First release	Traffic Server 3.0

INKIOBufferDestroy

Destroys an IO buffer.

Prototype	`INKReturnCode` **`INKIOBufferDestroy`** `(INKIOBuffer` *`bufp`*`)`
Description	Destroys the IO buffer *`bufp`*. Since two IO buffers can share data this does not necessarily free all of the data associated with the IO buffer but simply decrements the appropriate reference counts.
Returns	`INK_SUCCESS` if the IO buffer is successfully destroyed. `INK_ERROR` if an error occurs.
First release	Traffic Server 3.0

INKIOBufferProduce

Makes a specified number of bytes of data available for reading.

Prototype	`INKReturnCode` **`INKIOBufferProduce`** `(INKIObuffer` *`bufp`*`, int` *`nbytes`*`)`
Description	Makes *`nbytes`* of data available for reading in the buffer *`bufp`*. A common paradigm for writing to a buffer is to copy data into a buffer block and then call `INKIOBufferProduce` to make the new data visible to any readers.
Returns	`INK_SUCCESS` if the operation completes successfully. `INK_ERROR` if an error occurs.
First release	Traffic Server 3.0

INKIOBufferReaderAlloc

Allocates an IO buffer reader.

Prototype	`INKIOBufferReader` **`INKIOBufferReaderAlloc`** `(INKIOBuffer` *`bufp`*`)`
Description	Allocates an IO buffer reader for the IO buffer *`bufp`*.
Returns	A handle to the newly allocated IO buffer. `INK_ERROR_PTR` if an error occurs.
First release	Traffic Server 3.0

INKIOBufferReaderAvail

Gets the number of bytes available for reading.

Prototype	`int` **`INKIOBufferReaderAvail`** `(INKIOBufferReader` *`readerp`*`)`
Description	Gets the total number of bytes available for reading by the IO buffer reader *`readerp`*.

Returns	The number of bytes available for reading. `INK_ERROR` if an error occurs.
First release	Traffic Server 3.0

INKIOBufferReaderClone

Clones an IO buffer reader.

Prototype	`INKIOBufferReader` **`INKIOBufferReaderClone`** `(INKIOBufferReader` *`readerp`*`)`
Description	Makes a clone of the IO buffer reader *`readerp`*. The cloned reader will point to the same IO buffer and initially have the same read offset as *`readerp`*.
Returns	A handle to the cloned IO buffer. `INK_ERROR_PTR` if an error occurs.
First release	Traffic Server 3.0

INKIOBufferReaderConsume

Consumes an IO buffer reader.

Prototype	`INKReturnCode` **`INKIOBufferReaderConsume`** `(INKIOBufferReader` *`readerp`*`,` `int` *`nbytes`*`)`
Description	Moves the read offset for the IO buffer reader *`readerp`* ahead by *`nbytes`*. **Caution:** once a reader moves its offset ahead it can never move it back. When a reader moves its offset the data it has moved passed is potentially freed at that moment.
Returns	`INK_SUCCESS` if the operation completes successfully. `INK_ERROR` if an error occurs.
First release	Traffic Server 3.0

INKIOBufferReaderFree

Frees an IO buffer reader.

Prototype	`INKReturnCode` **`INKIOBufferReaderFree`** `(INKIOBufferReader` *`readerp`*`)`
Description	Frees an IO buffer reader. The IO buffer maintains a reference to each reader accessing it and will free those references when the buffer gets destroyed making it unnecessary to call `INKIOBufferReaderFree`. It is sometimes useful to free an IO buffer reader if the reader is no longer being used to allow the buffer data to automatically be de-allocated when other readers have consumed it.
Returns	`INK_SUCCESS` if the IO buffer is successfully freed. `INK_ERROR` if an error occurs.
First release	Traffic Server 3.0

INKIOBufferReaderStart

Starts an IO buffer reader.

Prototype	`INKIOBufferBlock` **`INKIOBufferReaderStart`** `(INKIOBufferReader` *`readerp`*`)`
Description	Gets the read start block for the IO buffer reader. `INKIOBufferReaderStart` may return `NULL` if there is no data available for reading. It may also return an IO buffer block with no data available for reading. Both conditions need to be checked for.
Returns	The read start block for the IO buffer reader. `INK_ERROR_PTR` if an error occurs.
First release	Traffic Server 3.0

INKIOBufferSizedCreate

Creates an `INKIOBuffer` with specified size index.

Prototype	`INKIOBuffer INKIOBufferSizedCreate (INKIOBufferSizeIndex` *`index`*`)`

Arguments	`INKIOBufferSizeIndex` *`index`* is the size of the new `IOBuffer` to create and should be one of the following values:
	`INK_IOBUFFER_SIZE_INDEX_128`
	`INK_IOBUFFER_SIZE_INDEX_256`
	`INK_IOBUFFER_SIZE_INDEX_512`
	`INK_IOBUFFER_SIZE_INDEX_1K`
	`INK_IOBUFFER_SIZE_INDEX_2K`
	`INK_IOBUFFER_SIZE_INDEX_4K`
	`INK_IOBUFFER_SIZE_INDEX_8K`
	`INK_IOBUFFER_SIZE_INDEX_16K`
	`INK_IOBUFFER_SIZE_INDEX_32K`

Description	Creates an `INKIOBuffer` of the specifed size.
Returns	An IOBuffer object if the API call is successful. `INK_ERROR_PTR` if an error occurs while calling the API or if an argument is invalid.
First release	Traffic Server 5.2

INKIOBufferStart

Starts an IO buffer.

Prototype	`INKIOBufferBlock` **`INKIOBufferStart`** `(INKIOBuffer` *`bufp`*`)`
Description	Gets the write start block for the IO buffer *`bufp`*. `INKIOBufferStart` will always return a block with some non-zero amount of space available for writing. A new block will be added if necessary to accomplish this.
Returns	The write start block for the IO buffer writer. `INK_ERROR_PTR` if an error occurs.
First release	Traffic Server 3.0

INKIOBufferWaterMarkGet

Gets the current watermark for the specified buffer.

Prototype	`InkReturnCode INKIOBufferWaterMarkGet (INKIOBuffer bufp, int *watermark)`
Arguments	`INKIOBuffer bufp` is the IOBuffer whose water_mark is to be obtained. `int *watermark` is set to the watermark value.
Description	Gets the current watermark for the specified buffer. A water mark applies only to a NetVConnection and should be used only when reading data from a NetVC. Note that this is only applicable for NetVC. When water mark is set to N, and after having called INKVConnRead, the Net processor calls back the reader (with an event INK_VCONN_READ_READY) only when at least N bytes of data are available for reading.
Returns	`INK_SUCCESS` if API call is successful. `INK_ERROR` if an error occurs while calling the API or if an argument is invalid.
First release	Traffic Server 5.2

INKIOBufferWaterMarkSet

Sets the current watermark for the specified buffer.

Prototype	`INKReturnCode INKIOBufferWaterMarkSet (INKIOBuffer bufp,int water_mark)`
Arguments	`INKIOBuffer bufp` is the `IOBuffer` whose watermark is to be set. `int water_mark` is the watermark value to set for `bufp`.
Description	Sets the current watermark of the specified buffer. A water mark applies only to a NetVConnection and should be used only when reading data from a NetVC. When water mark is set to N, and after having called INKVConnRead, the Net processor calls back the reader (with an event INK_VCONN_READ_READY) only when at least N bytes of data are available for reading.
Returns	`INK_SUCCESS` if the operation completes successfully. `INK_ERROR` if an error occurs while calling the API or if an argument is invalid.
First release	Traffic Server 5.2

INKIOBufferWrite

Appends the specified number of bytes from a buffer to the IO buffer.

Prototype	`int INKIOBufferWrite (INKIOBuffer bufp, const char *buf, int len)`
Arguments	`INKIOBuffer bufp` is the target `IOBuffer` to receive the data. `const char *buf` is the buffer which contains the data. `int len` is the length of the data to write.
Description	This function appends data from `*buf` to `IOBuffer bufp`, the length of data being appended is given in `len`. The returned value is the actual length of data being appended.

Returns The length of data copied if API call is successful.

INK_ERROR if an error occurs while calling the API or if an argument is invalid.

Example INKIOBufferWrite offers the same functionality as the deprecated functions INKIOBufferAppend, INKIOBufferDataCreate and INKIOBufferBlockCreate. To append the content of a buffer *buf* of size *len* into an IOBuffer, we recommend using INKIOBufferWrite which has the following prototype:

int INKIOBufferWrite (INKIOBuffer *bufp*, const char **buf*, int *len*);

The equivalent of this API in SDK2.0 is the following snippet of code:

```
 INKIOBufferBlock block;
 int avail, ndone, ntodo, towrite;
 char *ptr_block;

 ndone = 0;
 ntodo = len;
 while (ntodo > 0) {
      /* INKIOBufferStart allocates more blocks if required */
      block = INKIOBufferStart(bufp);
      ptr_block = INKIOBufferBlockWriteStart (block, &avail);
      towrite = min(ntodo, avail);
      memcpy (ptr_block, buf+ndone, towrite);
      INKIOBufferProduce(bufp, towrite);
      ntodo -= towrite;
      ndone += towrite;
}
```

First release Traffic Server 5.2

Management interface function

INKMgmtUpdateRegister

Sets up a plugin's management interface.

Prototype

```
INKReturnCode INKMgmtUpdateRegister (INKCont contp,
          const char *plugin_name, const char *path)
```

Arguments

contp is the continuation to be called back if the plugin's configuration is changed. The handler function for this continutation must handle the event INK_EVENT_MGMT_UPDATE.

plugin_name is the name of the plugin. This name must match the name of the plugin specified in your CGI form submission for INK_PLUGIN_NAME.

path is the location of the plugin's interface, relative to the Traffic Edge plugin directory (as specified in the records.config variable proxy.config.plugin.plugin_dir). If your plugin has a web user interface, then *path* must be located under the Traffic Edge config directory. This is because Traffic Manager derives the root of all of its web interfaces from the Traffic Edge config directory.

For example, *path* could be Blacklist/ui/index.html or Blacklist/ui/index.cgi.

The Traffic Edge administrator can view the interface at the following URL:

http://traffic_manager:8081/plugins/Blacklist/ui/index.html

Alternatively the administrator can access the interface in the Traffic Manager UI, through the **Plugin** icon in the **Configure** tab.

Description

Informs Traffic Manager about your plugin's interface (in the path argument).

Sets up a callback to your plugin when configuration changes are submitted. Your CGI program must set INK_PLUGIN_NAME to be the name of your plugin, so that Traffic Manager knows who to tell Traffic Edge to call. Traffic Edge calls back the continuation with the event INK_EVENT_MGMT_UPDATE. (The handler function for the continuation must handle the event INK_EVENT_MGMT_UPDATE.) See the *blacklist-1.c, on page 245* for an example.

Returns

INK_SUCCESS if the operation completes successfully.

INK_ERROR if an error occurs.

First release

Traffic Server 3.5

Traffic Edge Configuration Read Functions

INKMgmtCounterGet

Get a records.config variable of type counter.

Prototype

```
int INKMgmtCounterGet (const char *var_name, INKMgmtCounter *result)
```

Arguments

var_name is the name of the variable you want from records.config.

result is a pointer to the value of the variable. This value is of type INKMgmtCounter.

Description

INKMgmtCounterGet obtains the value of the specified records.config variable of type counter, and stores the value in *result*.

Returns If `INKMgmtCounterGet` could not get the variable, it returns zero. If successful, a nonzero value is returned.

First release Traffic Server 3.5

INKMgmtFloatGet

Get a `records.config` variable of type `float`.

Prototype `int` **`INKMgmtFloatGet`** `(const char *`*`var_name`*`, INKMgmtFloat *`*`result`*`)`

Arguments *`var_name`* is the name of the variable you want from `records.config`.
`result` is a pointer to the value of the variable. This value is of type `INKMgmtFloat`.

Description `INKMgmtFloatGet` obtains the value of the specified `records.config` variable of type `float`, and stores the value in *`result`*.

Returns If `INKMgmtFloatGet` could not get the variable, it returns zero. If it was successful, a nonzero value is returned.

First release Traffic Server 3.5

INKMgmtIntGet

Get a `records.config` variable of type `int`.

Prototype `int` **`INKMgmtIntGet`** `(const char *`*`var_name,`* `INKMgmtInt *`*`result`*`)`

Arguments *`var_name`* is the name of the variable you want from `records.config`.
`result` is a pointer to the value of the variable. This value is of type `INKMgmtInt`.

Description `INKMgmtIntGet` obtains the value of the specified `records.config` variable of type `int`, and stores the value in *`result`*.

Returns If `INKMgmtIntGet` could not get the variable, it returns zero. If it was successful, a nonzero value is returned.

Example The following code fragment does something if keepalive is enabled on Traffic Edge:

```
INKMgmtInt result;
if (INKMgmtIntGet(“proxy.config.http.keep_alive_enabled“, &result)) {
   if (result){
   // keepalive is enabled, do something
   }
}
else INKError (“could not retrieve value\n”);
```

First release Traffic Server 3.5

INKMgmtStringGet

Get a `records.config` variable of type `String`.

Prototype `int` **`INKMgmtStringGet`** `(const char *`*`var_name,`* `INKMgmtString *`*`result`*`)`

Arguments *`var_name`* is the name of the variable you want from `records.config`.
`result` is a pointer to the value of the variable. This value is of type `INKMgmtString`.

Description	`INKMgmtStringGet` obtains the value of the specified `records.config` variable of type `String`, and stores the value in *`result`*. When done with the result, your plugin must deallocate the result string with a call to *INKfree*.
Returns	If `INKMgmtStringGet` could not get the variable, it returns zero. If it was successful, a nonzero value is returned.
First release	Traffic Server 3.5

Customer installation and licensing functions

INKInstallDirGet

Gets Traffic Edge's install directory.

Prototype	`const char * INKInstallDirGet(void)`
Description	Get Traffic Edge's installation directory.
Returns	A pointer to a string containing the Traffic Edge's installation directory.
First release	Traffic Server 3.5

INKPluginDirGet

Gets the plugin directory.

Prototype	`const char * INKPluginDirGet(void)`
Description	Get the plugin directory relative to Traffic Edge's install directory. This path (relative to the Traffic Edge install directory) is stored in the `records.config` variable `proxy.config.plugin.plugin_dir` The default value is `config/plugin`.
Returns	A pointer to a string containing the plugin directory.
Example	To open the file `Blacklist/ui/blacklist_config.txt`, use `INKfopen ("INKInstallDirGet()/INKPluginDirGet()/Blacklist/ui/blacklist_config.txt");`
First release	Traffic Server 3.5

INKPluginLicenseRequired

Lets Traffic Edge know that a license key is required for the plugin.

Prototype	`int INKPluginLicenseRequired(void)`
Description	Determines if a license is required and, if so, Traffic Edge looks at the `plugin.db` file for the license key. If this function is not defined, a license is not required for the plugin.
Returns	Returns zero if no license is required. Returns 1 if a license is required.

Example

```
#include <stdio.h>
#include "InkAPI.h"

void INKPluginInit (int argc, const char *argv[])
{
    printf ("hello world\n");
}
int INKPluginLicenseRequired(void)
{
    return 1;
}
```

First release Traffic Server 3.5

Statistics functions

Uncoupled statistics

INKStatFloatGet

Obtains the value of a float stat.

Prototype `INKReturnCode` **`INKStatFloatGet`**`(INKStat` *`stat,`* `float` *`*value`*`)`

Returns `INK_SUCCESS` if the API is called successfully.
`INK_ERROR` if an error occurs while calling the API or if an argument is invalid.

First release Traffic Server 3.5

INKStatIntGet

Obtains the value of an integer stat.

Prototype `INKReturnCode` **`INKStatIntGet`**`(INKStat` *`stat,`* `INK64` *`*value`*`)`

Returns `INK_SUCCESS` if the API is called successfully.
`INK_ERROR` if an error occurs while calling the API or if an argument is invalid.

First release Traffic Server 3.5

INKStatFloatAddTo

Adds a float value to a float statistic.

Prototype `INKReturnCode` **`INKStatFloatAddTo`** `( INKStat` *`the_stat`*`, float` *`amount`*`)`

Description `Adds a float value to a float statistic.`

Returns	`INK_SUCCESS` if the operation completes successfully. `INK_ERROR` if an error occurs.
First release	Traffic Server 3.5

INKStatIntAddTo

Adds an INK64 value to an integer statistic.

Prototype	`INKReturnCode` **`INKStatIntAddTo`** `( INKStat` *`the_stat`*`, INK64` *`amount`*`)`
Description	`Adds an INK64 value to an integer statistic`
Returns	`INK_SUCCESS` if the operation completes successfully. `INK_ERROR` if an error occurs.
First release	Traffic Server 3.5

INKStatCreate

Creates a new `INKStat`.

Prototype	`INKStat` **`INKStatCreate`** `( const char *` *`the_name`*`, INKStatTypes` *`the_type`*`)`
Description	Creates a new INKStat. The value pointed to by*`the_name`* is the name you use to view the statistic using Traffic Line. See *Viewing statistics using Traffic Line, on page 139*. There are two `INKStatTypes`: `INKSTAT_TYPE_INT64`, and `INKSTAT_TYPE_FLOAT`.
Returns	A handle to the newly created `INKStat`. `INK_ERROR_PTR` if an error occurs.
First release	Traffic Server 3.5

INKStatDecrement

Decrements a stat.

Prototype	`INKReturnCode` **`INKStatDecrement`**`(INKStat` *`the_stat`* `)`
Description	`Decrements a stat.`
Returns	`INK_SUCCESS` if the operation completes successfully. `INK_ERROR` if an error occurs.
First release	Traffic Server 3.5

INKStatIncrement

Increments a stat.

Prototype	`INKReturnCode` **`INKStatIncrement`**`(INKStat` *`the_stat`* `)`
Description	`Increments a stat.`

Returns	INK_SUCCESS if the operation completes successfully. INK_ERROR if an error occurs.
First release	Traffic Server 3.5

INKStatFloatSet

Sets the value of a float stat to a particular value.

Prototype	`INKReturnCode` **`INKStatFloatSet`**`(INKStat` *`the_stat`* `, float` *`the_value`* `)`
Description	`Sets the value of a float stat to the specified value.`
Returns	INK_SUCCESS if the operation completes successfully. INK_ERROR if an error occurs.
First release	Traffic Server 3.5

INKStatIntSet

Sets the value of an integer stat to a particular value.

Prototype	`INKReturnCode` **`INKStatIntSet`**`(INKStat` *`the_stat`* `, INK64` *`the_value`* `)`
Description	`Sets the value of a integer stat to a particular value.`
Returns	INK_SUCCESS if the operation completes successfully. INK_ERROR if an error occurs.
First release	Traffic Server 3.5

Coupled statistics

INKStatCoupledGlobalAdd

.Creates a global coupled stat.

Prototype	`INKStat` **`INKStatCoupledGlobalAdd`** `(INKCoupledStat` *`global_copy`* `,` `const char *` *`the_name`* `, INKStatTypes` *`the_type`* `)`
Description	*`global_copy`* is the name of the global coupled stat category to which your new coupled stat belongs. *`the_name`* is the name you use to view the statistic using Traffic Line. See *Viewing statistics using Traffic Line, on page 139*. There are two `INKStatTypes`: `INKSTAT_TYPE_INT64`, and `INKSTAT_TYPE_FLOAT`. See *To add coupled statistics:, on page 138*.
Returns	A handle to the newly created global coupled stat. INK_ERROR_PTR if an error occurs.
First release	Traffic Server 3.5

INKStatCoupledLocalAdd

Creates a local copy of a global coupled stat.

Prototype	`INKStat` **`INKStatCoupledLocalAdd`** `(INKCoupledStat` *`local_copy`* `,` `const char *` *`the_name`* `,` `INKStatTypes` *`the_type`* `)`
Description	*`lcoal_copy`* is the name of the local coupled stat category to which your new coupled stat belongs. *`the_name`* is the name you use to view the statistic using Traffic Line. See *Viewing statistics using Traffic Line, on page 139*. There are two `INKStatTypes`: `INKSTAT_TYPE_INT64`, and `INKSTAT_TYPE_FLOAT`. See *To add coupled statistics:, on page 138*.
Returns	A handle to a local copy of the global coupled stat. `INK_ERROR_PTR` if an error occurs.
First release	Traffic Server 3.5

INKStatCoupledGlobalCategoryCreate

Creates a global coupled stat category.

Prototype	`INKCoupledStat` **`INKStatCoupledGlobalCategoryCreate`** `(` `const char *` *`the_name`* `)`
Description	Returns a new global coupled stat category. Use this function in `INKPluginInit`. The name argument is the name you use to access this stat in Traffic Line. See *Viewing statistics using Traffic Line, on page 139*. See *To add coupled statistics:, on page 138*.
Returns	A handle to a the newly created global coupled stat category. `INK_ERROR_PTR` if an error occurs.
First release	Traffic Server 3.5

INKStatCoupledLocalCopyCreate

.Creates a local copy of a global coupled stat category.

Prototype	`INKCoupledStat` **`INKStatCoupledLocalCopyCreate`** `( const char *` *`the_name`* `,` `INKCoupledStat` *`global_copy`*`)`
Description	Returns a new local coupled stat category. Use this function in any routine where you need to modify local copies of global statistics. The name argument is the name you use to access this stat in Traffic Line. See *Viewing statistics using Traffic Line, on page 139*. See *To add coupled statistics:, on page 138*.
Returns	A handle to the local copy of the global coupled stat category. `INK_ERROR_PTR` if an error occurs.
First release	Traffic Server 3.5

INKStatCoupledLocalCopyDestroy

.Destroys a local category of statistics.

Prototype	`INKReturnCode` **`INKStatCoupledLocalCopyDestroy`** `(` `INKCoupledStat` *`local_copy`*`)`
Description	Destroys a local statistics category. Always destroy the local category when you are done with it. See *To add coupled statistics:, on page 138*.
Returns	`INK_SUCCESS` if the operation completes successfully. `INK_ERROR` if an error occurs.
First release	Traffic Server 3.5

INKStatsCoupledUpdate

Updates a category of coupled statistics.

Prototype	`INKReturnCode` **`INKStatsCoupledUpdate (`**`INKCoupledStat` *`local_copy`*`)`
Description	Updates all of the coupled stats belonging to the category *`local_copy`*. See *To add coupled statistics:, on page 138*.
Returns	`INK_SUCCESS` if the operation completes successfully. `INK_ERROR` if an error occurs.
First release	Traffic Server 3.5

Logging functions

INKTextLogObjectCreate

Creates a new custom log for your plugin.

Prototype	`INKReturnCode` **`INKTextLogObjectCreate`** `(const char *`*`filename`*`, int` *`mode`*`,` `INKTextLogObject *`*`new_logobj`*`)`

Arguments	`Const char *`*`filename`* is the name of the new log file. The new log file is created in the log directory. You can specify a path to a subdirectory within the log directory (e.g. `subdir/filename`) but make sure you create the subdirectory first. If you do not specify a file name extension, the extension `.log` is automatically added.
	The logs you create are treated like ordinary logs; they are rolled if log rolling is enabled. (Log collation is not supported though).
	`int` *`mode`* is one (or both) of the following (can be 0):
	`INK_LOG_MODE_ADD_TIMESTAMP`
	Whenever the plugin makes a log entry using `INKTextLogObjectWrite` (see below), it prepends the entry with a timestamp.
	`INK_LOG_MODE_DO_NOT_RENAME`
	This means that if there is a filename conflict, Traffic Edge should not attempt to rename the custom log. The consequence of a name conflict is that the custom log is not created.
	`INKTextLogObject *`*`new_logobj`* is set to the newly created log object.

Description Creates a custom log for your plugin. Once log object is created, APIs `INKTextLogObjectRollingEnabledSet`, `INKTextLogObjectRollingIntervalSecSet`, `INKTExtLogObjectRollingOffsetHrSet` can be used on it to set properties.

If the value of mode is not a valid value, then the behavior of the API cannot be predicted.

Returns `INK_SUCCESS` if API is called successfully.

`INK_ERROR` if an error occurs while calling the API or if an argument is invalid.

Example Example: suppose you call

```
INKTextLogObjectCreate ("squid" , mode, NULL, &log);
```

If *mode* is `INK_LOG_MODE_DO_NOT_RENAME`, you will NOT get a new log (you'll get an error) if squid.log already exists.

If *mode* is not `INK_LOG_MODE_DO_NOT_RENAME`, Traffic Edge tries to rename the log to a new name (it will try `squid_1.log`).

If a log object is created with `INK_LOG_MODE_DO_NOT_RENAME` mode and a log with the same file name pre-exists, then the signature (type of log file) is compared. If the signature log files match, the pre-existing file is opened and logging is resumed at the end of the file. IF the signatures do not match, an error is returned.

If a log object is created without `INK_LOG_MODE_DO_NOT_RENAME` mode and a log with the same file name pre-exists, then the signature (type of log file) is compared. If the signatures of the log files match, the pre-existing file is opened and logging is resumed at the end of the file. If the signature does not match, another file with `filename_1.log` is tried and so on.

Signature of log file is a type of log file. Log files can be structured/fixed format log files or unstructured/free format log files. All free format log files have the same signature, while structure log files have the structure/fixed format of the log file as its signature.

First Release Traffic Server 5.2

INKTextLogObjectHeaderSet

Sets a log file header.

Prototype

```
INKReturnCode INKTextLogObjectHeaderSet ( INKTextLogObject the_object,
   const char *header)
```

Arguments `INKTextLogObject` *`the_object`* is the log object you want to set the header.

`const char *`*`header`* is a log file header.

Description A header for a log object is the banner (a text line) which is printed at the top of the log file. This API must be used once the object is created (using `INKTextLogObjectCreate`) and before writing into logs (using `INKTextLogObjectWrite`). By default a null header (empty line) is used.

Returns `INK_SUCCESS` if API is called successfully.

`INK_ERROR` if an error occurs while calling the API or if an argument is invalid.

First Release Traffic Server 5.2

INKTextLogObjectRollingEnabledSet

Enable/disable rolling for a log object..

Prototype `INKReturnCode` **`INKTextLogObjectRollingEnabledSet`** `( INKTextLogObject` *`the_object,`* `int` *`*rolling_enabled`*`)`

Arguments `INKTextLogObject` *`the_object`* is the log object you want to enable/disable rolling.

`int` *`rolling_enabled`* 1 to enable rolling, 0 to disable.

Description This API must be used once the object is created (using `INKTextLogObjectCreate`) and before writing into logs (using `INKTextLogObjectWrite`). If `INKTextLogObjectRollingEnabledSet` is not called, the default value as specified in `records.config` by parameter `proxy.config.log2.rolling_enabled` is used.

The rolling interval and offset can be specified using the APIs `INKTextLogObjectRollingIntervalSecSet` and `INKTextLogObjectRollingOffsetHrSet`.

Returns `INK_SUCCESS` if API is called successfully.

`INK_ERROR` if an error occurs while calling the API or if an argument is invalid.

Example Rolling example:

If rolling is enabled, the rolling interval set to 21600 sec (6 hours) and the offset hour set to 0 (midnight). Then the logs will be rolled at 0:00am, 06:00am, 12:00pm and 18:00pm each day.

Note: If the maximum amount of disk space reserved for logs is exhausted and if parameter `proxy.config.log2.auto_delete_rolled_files` is enabled in `records.config`, rolled files are automatically deleted by Traffic Edge to free up some space.

First Release Traffic Server 5.2

INKTextLogObjectRollingIntervalSecSet

Sets the rolling interval for a log object.

Prototype `INKReturnCode` **`INKTextLogObjectRollingIntervalSecSet`** `( INKTextLogObject` *`the_object,`* `int` *`rolling_interval_sec`*`)`

Arguments `INKTextLogObject` *`the_object`* is the log object you want to set the rolling interval.

`int` *`rolling_interval_sec`* is the rolling interval, in seconds.

Description This API must be used once the object is created (using `INKTextLogObjectCreate`) and before writing into logs (using `INKTextLogObjectWrite`). By default a null header is used.

If `INKTextLogObjectRollingIntervalSecSet` is not called, the defaut value as specified in `records.config` by parameter `proxy.config.log2.rolling_interval_sec` is used.

The rolling offset can be specified using the API `INKTextLogObjectRollingOffsetHrSet`.

Returns `INK_SUCCESS` if API is called successfully.

`INK_ERROR` if an error occurs while calling the API or if an argument is invalid.

First Release Traffic Server 5.2

INKTextLogObjectRollingOffsetHrSet

Sets Set the rolling offset for a log object.

Prototype `INKReturnCode` **`INKTextLogObjectRollingOffsetHrSet`** `( INKTextLogObject` *`the_object,`* `int` *`rolling_offset_hr`*`)`

Arguments `INKTextLogObject` *`the_object`* is the log object you want to set the rolling offset.

`int` *`rolling_offset_hr`* is the rolling interval, in seconds.

Description This API must be used once the object is created (using `INKTextLogObjectCreate`) and before writing into logs (using `INKTextLogObjectWrite`). By default a null header is used.

If `INKTextLogObjectRollingOffsetHrSet` is not called, the defaut value as specified in `records.config` by parameter `proxy.config.log2.rolling_offset_hr` is used.

The rolling interval can be specified using the API `INKTextLogObjectRollingIntervalSecSet`.

Returns `INK_SUCCESS` if API is called successfully.

`INK_ERROR` if an error occurs while calling the API or if an argument is invalid.

First Release Traffic Server 5.2

INKTextLogObjectWrite

Writes a text entry to a custom log file.

Prototype `InkReturnCode` **`INKTextLogObjectWrite`** `(INKTextLogObject` *`the_object`*`, char` *`*format`*`, ...)`

Arguments *`the_object`* is the log object to write to. You must first create this log file with `INKTextLogObjectCreate`.

`char` *`*format`* is a printf-style formatted statement to be printed.

`...` are the parameters in the formatted statement. A newline is automatically added to the end.

Description Writes a text entry to a custom log file.

Returns `INK_SUCCESS` if API is called successfully.

`INK_ERROR` if an error occurs while calling the API or if an argument is invalid.

Example Suppose you call:

`int` *`my_value = 2001`*;

`INKTextLogObjectWrite (log, “my value: %d”, my_value);`

If mode is set to `ADD_TIMESTAMP`, the log should look like:

`<timestamp> my value: 2001`

First Release Traffic Server 5.2

INKTextLogObjectFlush

Flushes the contents of a specified log file’s log write buffer to disk.

Prototype `INKReturnCode` **`INKTextLogObjectFlush`** `(INKTextLogObject` *`the_object`*`)`

Arguments `INKTextLogObject` *`the_object`* is the log file whose write buffer you want to flush. You have to first create this object with `INKTextLogObjectCreate`.

Description	This immediately flushes the contents of the log write buffer for `the_object` to disk. Use this call only if you want to make sure that log entries are flushed immediately. This call has a performance cost. Traffic Edge flushes the log buffer automatically about every 1 second.
Returns	`INK_SUCCESS` if the API is called successfully. `INK_ERROR` if an error occurs while calling the API or if an argument is invalid.
First Release	Traffic Server 5.2

INKTextLogObjectDestroy

Destroys a custom log file created by `INKTextLogObjectCreate`.

Prototype	`INKReturnCode` **`INKTextLogObjectDestroy`** `(INKTextLogObject` *`the_object`*`)`
Arguments	`INKTextLogObject` *`the_object`* is the custom log file you want to destroy. You have to first create this object with `INKTextLogObjectCreate`.
Description	Destroys a log object (a plugin's custom log file) and releases the memory allocated to it. Use this call if done with the log.
Returns	`INK_SUCCESS` if the API is called successfully. `INK_ERROR` if an error occurs while calling the API or if an argument is invalid.
First Release	Traffic Server 5.2

APPENDIX A

Sample Source Code

This appendix provides several source code examples. In the PDF and HTML formats of this book, function calls are linked to their references in the previous chapters. The following sample plugins are provided:

blacklist-1.c

The sample blacklisting plugin included in the Traffic Edge SDK is `blacklist-1.c`. This plugin checks every incoming HTTP client request against a list of blacklisted web sites. If the client requests a blacklisted site, the plugin returns an “access forbidden” message to the client.

This plugin illustrates:

- An HTTP transaction extension
- How to examine HTTP request headers
- How to use the logging interface
- How to use the plugin configuration management interface

```
/* blacklist-1.c:  an example program that denies client access
 *                 to blacklisted sites. This plugin illustrates
 *                 how to use configuration information from a
 *                 configuration file (blacklist.txt) that can be
 *                 updated through the Traffic Manager UI.
 *
 *   Copyright (c) 1999/2000 Inktomi Corporation.  All Rights Reserved.
 *   Authorized possession and use of this software is only pursuant
 *   to the terms of a written license agreement.
 *
 *  Usage:
 *  (NT) : BlackList.dll
 *  (Solaris) : blacklist-1.so
 *
 *
 */

#include <stdio.h>
#include <string.h>
```

```
#include "InkAPI.h"

#define MAX_NSITES 500

static char* sites[MAX_NSITES];
static int nsites;
static INKMutex sites_mutex;
static INKTextLogObject log;

static void
handle_dns (INKHttpTxn txnp, INKCont contp)
{
    INKMBuffer bufp;
    INKMLoc hdr_loc;
    INKMLoc url_loc;
    const char *host;
    int i;
    int host_length;

    if (!INKHttpTxnClientReqGet (txnp, &bufp, &hdr_loc)) {
        INKError ("couldn't retrieve client request header\n");
        goto done;
    }

    url_loc = INKHttpHdrUrlGet (bufp, hdr_loc);
    if (!url_loc) {
        INKError ("couldn't retrieve request url\n");
        INKHandleMLocRelease (bufp, INK_NULL_MLOC, hdr_loc);
        goto done;
    }

    host = INKUrlHostGet (bufp, url_loc, &host_length);
    if (!host) {
        INKError ("couldn't retrieve request hostname\n");
        INKHandleMLocRelease (bufp, hdr_loc, url_loc);
        INKHandleMLocRelease (bufp, INK_NULL_MLOC, hdr_loc);
        goto done;
    }

    INKMutexLock(sites_mutex);

    for (i = 0; i < nsites; i++) {
        if (strncmp (host, sites[i], host_length) == 0) {
        if (log) {
        INKTextLogObjectWrite(log, "blacklisting site: %s", sites[i]);
```

```
        } else {
        printf ("blacklisting site: %s\n", sites[i]);
        }
            INKHttpTxnHookAdd (txnp,
                  INK_HTTP_SEND_RESPONSE_HDR_HOOK,
                  contp);
            INKHandleStringRelease (bufp, url_loc, host);
            INKHandleMLocRelease (bufp, hdr_loc, url_loc);
            INKHandleMLocRelease (bufp, INK_NULL_MLOC, hdr_loc);
            INKHttpTxnReenable (txnp, INK_EVENT_HTTP_ERROR);
        INKMutexUnlock(sites_mutex);
            return;
        }
    }

    INKMutexUnlock(sites_mutex);
    INKHandleStringRelease (bufp, url_loc, host);
    INKHandleMLocRelease (bufp, hdr_loc, url_loc);
    INKHandleMLocRelease (bufp, INK_NULL_MLOC, hdr_loc);

 done:
    INKHttpTxnReenable (txnp, INK_EVENT_HTTP_CONTINUE);
}

static void
handle_response (INKHttpTxn txnp)
{
    INKMBuffer bufp;
    INKMLoc hdr_loc;
    INKMLoc url_loc;
    char *url_str;
    char *buf;
    int url_length;

    if (!INKHttpTxnClientRespGet (txnp, &bufp, &hdr_loc)) {
        INKError ("couldn't retrieve client response header\n");
        goto done;
    }

    INKHttpHdrStatusSet (bufp, hdr_loc, INK_HTTP_STATUS_FORBIDDEN);
    INKHttpHdrReasonSet (bufp, hdr_loc,
        INKHttpHdrReasonLookup (INK_HTTP_STATUS_FORBIDDEN),
        strlen (INKHttpHdrReasonLookup (INK_HTTP_STATUS_FORBIDDEN)) );

    if (!INKHttpTxnClientReqGet (txnp, &bufp, &hdr_loc)) {
        INKError ("couldn't retrieve client request header\n");
```

```
        INKHandleMLocRelease (bufp, INK_NULL_MLOC, hdr_loc);
        goto done;
    }

    url_loc = INKHttpHdrUrlGet (bufp, hdr_loc);
    if (!url_loc) {
        INKError ("couldn't retrieve request url\n");
        INKHandleMLocRelease (bufp, INK_NULL_MLOC, hdr_loc);
        goto done;
    }

    buf = (char *)INKmalloc (4096);

    url_str = INKUrlStringGet (bufp, url_loc, &url_length);
    sprintf (buf, "You are forbidden from accessing \"%s\"\n", url_str);
    INKfree (url_str);
    INKHandleMLocRelease (bufp, hdr_loc, url_loc);
    INKHandleMLocRelease (bufp, INK_NULL_MLOC, hdr_loc);

    INKHttpTxnErrorBodySet (txnp, buf, strlen (buf), NULL);

 done:
    INKHttpTxnReenable (txnp, INK_EVENT_HTTP_CONTINUE);
}

static void
read_blacklist (void)
{
    char blacklist_file[1024];
    INKFile file;

    sprintf (blacklist_file, "%s/blacklist.txt", INKPluginDirGet());
    file = INKfopen(blacklist_file, "r");

    INKMutexLock (sites_mutex);
    nsites = 0;

    if (file != NULL) {
    char buffer[1024];

    while (INKfgets (file, buffer, sizeof(buffer)-1) != NULL &&
           nsites < MAX_NSITES) {
        char* eol;
        if ((eol = strstr(buffer, "\r\n")) != NULL) {
        /* To handle newlines on Windows */
```

```
        *eol = '\0';
        } else if ((eol = strchr(buffer, '\n')) != NULL) {
        *eol = '\0';
        } else {
        /* Not a valid line, skip it */
        continue;
        }
        if (sites[nsites] != NULL) {
        INKfree (sites[nsites]);
        }
        sites[nsites] = INKstrdup (buffer);
        nsites++;
    }

    INKfclose (file);
    } else {
    INKError ("unable to open %s\n", blacklist_file);
    INKError ("all sites will be allowed\n", blacklist_file);
    }

    INKMutexUnlock (sites_mutex);

}

static int
blacklist_plugin (INKCont contp, INKEvent event, void *edata)
{
    INKHttpTxn txnp = (INKHttpTxn) edata;

    switch (event) {
    case INK_EVENT_HTTP_OS_DNS:
        handle_dns (txnp, contp);
        return 0;
    case INK_EVENT_HTTP_SEND_RESPONSE_HDR:
        handle_response (txnp);
        return 0;
    case INK_EVENT_MGMT_UPDATE:
    read_blacklist ();
    return 0;
    default:
        break;
    }
    return 0;
}

int
```

```
check_ts_version() {

   const char* ts_version = INKTrafficServerVersionGet();
   int result = 0;

   if (ts_version) {
       int major_ts_version = 0;
       int minor_ts_version = 0;
       int patch_ts_version = 0;

       if (sscanf(ts_version, "%d.%d.%d", &major_ts_version,
          &minor_ts_version, &patch_ts_version) != 3) {
       return 0;
       }

       /* Since this is an TS-SDK 2.0 plugin, we need at
      least Traffic Server 3.5.2 to run */
       if (major_ts_version > 3) {
       result = 1;
       } else if (major_ts_version == 3) {
       if (minor_ts_version > 5) {
           result = 1;
       } else if (minor_ts_version == 5) {
           if (patch_ts_version >= 2) {
           result = 1;
           }
       }
       }
   }

   return result;
}

void
INKPluginInit (int argc, const char *argv[])
{
    int i;
    INKCont contp;
    INKPluginRegistrationInfo info;
    int error;

    info.plugin_name = "blacklist-1";
    info.vendor_name = "MyCompany";
    info.support_email = "ts-api-support@MyCompany.com";
```

```
    if (!INKPluginRegister (INK_SDK_VERSION_2_0 , &info)) {
        INKError ("Plugin registration failed.\n");
    }

    if (!check_ts_version()) {
    INKError ("Plugin requires Traffic Server 3.5.2 or later\n");
    return;
    }

    /* create an INKTextLogObject to log blacklisted requests to */
    log = INKTextLogObjectCreate("blacklist", INK_LOG_MODE_ADD_TIMESTAMP,
                NULL, &error);
    if (!log) {
    printf("Blacklist plugin: error %d while creating log\n", error);
    }

    sites_mutex = INKMutexCreate ();

    nsites = 0;
    for (i = 0; i < MAX_NSITES; i++) {
    sites[i] = NULL;
    }

    read_blacklist ();

    contp = INKContCreate (blacklist_plugin, NULL);

    INKHttpHookAdd (INK_HTTP_OS_DNS_HOOK, contp);

    INKMgmtUpdateRegister (contp, "Inktomi Blacklist Plugin", "blacklist.cgi");
}
```

APPENDIX B Deprecated Functions

This appendix lists the functions that are deprecated in SDK 5.2 and newer.

Deprecated MIME header functions

The following MIME field functions are deprecated in SDK 3.0.

INKMimeFieldCopy

Copies a MIME field from one location to another.

Prototype	`void` **`INKMimeFieldCopy`** `(INKMBuffer` *`dest_bufp`*`,INKMLoc` *`dest_offset`*`,INKMBuffer` *`src_bufp`*`,INKMLoc` *`src_offset`*`)`
Description	Copies the contents of the MIME field located at *`src_offset`* within the marshal buffer *`src_bufp`* to the MIME field located at *`dest_offset`* within the marshal buffer *`dest_bufp`*. `INKMimeFieldCopy` works correctly even if *`src_bufp`* and *`dest_bufp`* point to different marshal buffers. **Note**: you must first create the destination MIME field before copying into it.
First release	Traffic Server 3.0

INKMimeFieldCopyValues

Copies MIME field values from one location to another.

Prototype	`void` **`INKMimeFieldCopyValues`** `(INKMBuffer` *`dest_bufp`*`,INKMLoc` *`dest_offset`*`,INKMBuffer` *`src_bufp`*`,INKMLoc` *`src_offset`*`)`
Description	Copies the values contained within the MIME field located at *`src_offset`* within the marshal buffer *`src_bufp`* to the MIME field located at *`dest_offset`* within the marshal buffer *`dest_bufp`*. `INKMimeFieldCopyValues` works correctly even if *`src_bufp`* and *`dest_bufp`* point to different marshal buffers. `INKMIMEFieldCopyValues` does not copy the field's name.
First release	Traffic Server 3.0

INKMimeFieldCreate

Creates a new MIME field within a specified marshal buffer.

Prototype	`INKMLoc` **`INKMimeFieldCreate`** `(INKMBuffer` *`bufp`*`)`
Description	Creates a new MIME field with the marshal buffer *`bufp`*. Returns the offset location of the new MIME field. Release the created `INKMLoc` with a call to `INKHandleMLocRelease`.
First release	Traffic Server 3.0

INKMimeFieldDestroy

Deletes a specified MIME field from a marshal buffer.

Prototype	`void` **`INKMimeFieldDestroy`** `(INKMBuffer` *`bufp`*`,INKMLoc` *`offset`*`)`
Description	Destroys the MIME field located at *`offset`* within the marshal buffer *`bufp`*. Release the handle with a call to INKHandleMLocRelease.
First release	Traffic Server 3.0

INKMimeFieldLengthGet

Calculates the length of a string representation of a specified MIME field.

Prototype	`int` **`INKMimeFieldLengthGet`** `(INKMBuffer` *`bufp`*`,INKMLoc` *`offset`*`)`
Description	Calculates the length of the MIME field located at *`offset`* within the marshal buffer *`bufp`* if it were returned as a string. This is the length of the MIME field in its unparsed form.
First release	Traffic Server 3.0

INKMimeFieldNameGet

Gets the name and length of a specified MIME field.

Prototype	`const char*` **`INKMimeFieldNameGet`** `(INKMBuffer` *`bufp`*`,INKMLoc` *`offset`*`, int *`*`length`*`)`
Description	Returns the name of the field located at *`offset`* within the marshal buffer *`bufp`*. `INKMimeFieldNameGet` places the length of the returned string in the *`length`* argument. If *`length`* is NULL then no attempt is made to de-reference it. Release the returned string with a call to `INKHandleStringRelease`.
First release	Traffic Server 3.0

INKMimeFieldNameSet

Sets a specified MIME field's name.

Prototype	`void` **`INKMimeFieldNameSet`** `(INKMBuffer` *`bufp`*`,INKMLoc` *`offset`*`, const char *`*`name`*`, int` *`length`*`)`
Description	Sets the name of the field located at *`offset`* within the marshal buffer *`bufp`* to the string *`name`*. If *`length`* is `-1` then `INKMimeFieldNameSet` assumes that*`name`* is null-terminated. Otherwise, the length of the string *`name`* is taken to be *`length`*. `INKMimeFieldNameSet` copies the string to within *`bufp`*, so it is okay to modify or delete *`name`* after calling `INKMimeFieldNameSet`.
First release	Traffic Server 3.0

INKMimeFieldNext

Returns the next MIME field after a specified MIME field in a MIME header.

Prototype	`INKMLoc` **`INKMimeFieldNext`** `(INKMBuffer` *`bufp`*`,INKMLoc` *`offset`*`)`
Description	Conceptually, there are a list of MIME fields in a MIME header (see *"About HTTP headers" on page 83*). `INKMimeFieldNext` returns the location of the next field in the list after the field located at *`offset`* within the marshal buffer *`bufp`*. Release the returned `INKMLoc` with a call to `INKHandleMLocRelease`.
First release	Traffic Server 3.0

INKMimeFieldValueAppend

Appends a string to a specified value in a MIME field.

Prototype	`void` **`INKMimeFieldValueAppend`** `(INKMBuffer` *`bufp`*`, INKMLoc` *`offset`*`, int` *`idx`*`, const char *`*`value`*`, int` *`length`*`)`
Arguments	*`bufp`* is the marshal buffer containing the MIME field. *`offset`* is the location of the MIME field within the marshal buffer *`bufp`*. *`idx`* is the index of the field value to be appended. For example, in the MIME field `Foo: bar, car` the index of the value `bar` is `0`, and the index of `car` is `1`. *`value`* is the string to be appended to the MIME field value at *`idx`*. *`length`* is the length of the string *`value`* to be appended.
Description	Appends the string stored in *`value`* to a specific value in the MIME field located at *`offset`* within the marshal buffer *`bufp`*. The effect of `INKMimeFieldValueAppend` is as if the previous value were retrieved, the string *`value`* were appended to it and this new string were stored back in the MIME field at the same position. The *`idx`* parameter specifies which value in the field to append to. If *`idx`* is not between `0` and `INKMimeFieldValuesCount` (*`bufp`*, *`offset`*) `- 1` then no operation will be performed.
First release	Traffic Server 3.0

INKMimeFieldValueDelete

Deletes a specified value from a MIME field.

Prototype	`void` **`INKMimeFieldValueDelete`** `(INKMBuffer` *`bufp`*`,INKMLoc` *`offset`*`, int` *`idx`*`)`
Description	Removes and deletes a value from the MIME field located at *`offset`* within the marshal buffer *`bufp`*. The *`idx`* parameter specifies which value should be deleted. If *`idx`* is not between `0` and `INKMimeFieldValuesCount` (*`bufp`*, *`offset`*) `- 1` then no operation will be performed. Release the handle *`offset`* with a call to `INKHandleMLocRelease`.
First release	Traffic Server 3.0

INKMimeFieldValueGet

Gets a specified field value from a MIME header.

Prototype `const char* INKMimeFieldValueGet (INKMBuffer bufp, INKMLoc offset, int idx, int *length)`

Description Retrieves a string value from within the MIME field located at `offset` within the marshal buffer `bufp`. The `idx` parameter specifies which field to retrieve. The fields are numbered from `0` to `INKMimeFieldValuesCount` (`bufp`, `offset`) `- 1`. If `idx` does not lie within that range then `NULL` will be returned. The length of the returned string is placed in the `length` argument. If `length` is NULL then no attempt is made to dereference it.

Release the returned string with a call to `INKHandleStringRelease`.

First release Traffic Server 3.0

INKMimeFieldValueGetDate

Gets date value from a MIME field.

Prototype `time_t INKMimeFieldValueGetDate (INKMBuffer bufp, INKMLoc offset, int idx)`

Description Retrieves a date value from within the MIME field located at `offset` within the marshal buffer `bufp`. The `idx` parameter specifies which field to retrieve. The fields are numbered from `0` to `INKMimeFieldValuesCount` (`bufp`, `offset`) `- 1`. If `idx` does not lie within that range, `INKMimeFieldValueGetDate` returns `(time_t) 0`. All values are stored as strings within the MIME field. `INKMimeFieldValueGetDate` parses the string value to return an integer date representation.

First release Traffic Server 3.0

INKMimeFieldValueGetInt

Gets an integer field value in a MIME field.

Prototype `int INKMimeFieldValueGetInt (INKMBuffer bufp, INKMLoc offset, int idx)`

Description Retrieves an integer value from within the MIME field located at `offset` within the marshal buffer `bufp`. The `idx` parameter specifies which value within the field to retrieve. The fields are numbered from `0` to `INKMimeFieldValuesCount` (`bufp`, `offset`) `- 1`. If `idx` does not lie within that range, `INKMimeFieldValueGetInt` returns `(int) 0`. All values are stored as strings within the MIME field. `INKMimeFieldValueGetInt` parses the string value to return an integer.

First release Traffic Server 3.0

INKMimeFieldValueGetUint

Gets unsigned integer field value in a MIME field.

Prototype `unsigned int INKMimeFieldValueGetUint (INKMBuffer bufp, INKMLoc offset, int idx)`

Description Retrieves an unsigned integer value from within the MIME field located at *offset* within the marshal buffer *bufp*. The *idx* parameter specifies which field to retrieve. The fields are numbered from 0 to INKMimeFieldValuesCount (*bufp*, *offset*) - 1. If *idx* does not lie within that range, INKMimeFieldValueGetUnit returns (unsigned int) 0. All values are stored as strings within the MIME field. INKMimeFieldValueGetUint parses the string value to return an unsigned integer.

First release Traffic Server 3.0

INKMimeFieldValueInsert

Inserts a value into a specified location within a MIME field.

Prototype
```
INKMLoc INKMimeFieldValueInsert (INKMBuffer bufp,INKMLoc offset, const
    char *value, int length, int idx)
```

Description Inserts the string *value* into the MIME field located at *offset* within the marshal buffer *bufp*. If *length* is -1 then INKMimeFieldValueInsert assumes that *value* is null-terminated. Otherwise, the length of the string *value* is taken to be *length*. INKMimeFieldValueInsert copies the string to within *bufp*, so it is okay to modify or delete *value* after calling INKMimeFieldValueSet. The *idx* parameter specifies where the inserted value should be put with respect to the other values already in the MIME field. If *idx* is 0 then INKMimeFieldValueInsert prepends the value to the list of values in the field. Increasing values of *idx* place the value further down the list of values. If *idx* is -1, INKMimeFieldValueInsert appends the value to the list of values. Normal usage is to specify -1 for *idx* so that the value is appended to the list of values.

Release the returned INKMLoc with a call to INKHandleMLocRelease.

First release Traffic Server 3.0

INKMimeFieldValueInsertDate

Inserts a date value into a MIME field.

Prototype
```
INKMLoc INKMimeFieldValueInsertDate (INKMBuffer bufp,INKMLoc offset,
    time_t value, int idx)
```

Description Inserts the date *value* into the MIME field located at *offset* within the marshal buffer *bufp*. The *idx* parameter specifies where the inserted value should be put with respect to the other values already in the MIME field. If *idx* is 0 then the value is prepended to the list of values in the field. Increasing values of *idx* places the value further down the list of values. If *idx* is -1 then the value is appended to the list of values. Normal usage is to specify -1 for *idx* so that the value is appended to the list of values. All values are stored as strings within the MIME field. INKMimeFieldValueInsertDate simply formats the date into a string and then calls INKMimeFieldValueInsert.

Release the returned INKMLoc with a call to INKHandleMLocRelease.

First release Traffic Server 3.0

INKMimeFieldValueInsertInt

Inserts an integer value into a MIME field.

Prototype `INKMLoc` **`INKMimeFieldValueInsertInt`** `(INKMBuffer` *`bufp`*`,INKMLoc` *`offset`*`, int` *`value`*`, int` *`idx`*`)`

Description Inserts the integer *`value`* into the MIME field located at *`offset`* within the marshal buffer *`bufp`*. The *`idx`* parameter specifies where the inserted value should be put with respect to the other values already in the MIME field. If *`idx`* is `0` then the value is prepended to the list of values in the field. Increasing values of *`idx`* places the value further down the list of values. If *`idx`* is `-1` then the value is appended to the list of values. Normal usage is to specify `-1` for *`idx`* so that the value is appended to the list of values. All values are stored as strings within the MIME field. `INKMimeFieldValueInsertInt` simply formats the integer into a string and then calls `INKMimeFieldValueInsert`.

Release the returned `INKMLoc` with a call to `INKHandleMLocRelease`.

First release Traffic Server 3.0

INKMimeFieldValueInsertUint

Inserts an unsigned integer value into a MIME field.

Prototype `INKMLoc` **`INKMimeFieldValueInsertUint`** `(INKMBuffer` *`bufp`*`,INKMLoc` *`offset`*`, unsigned int` *`value`*`, int` *`idx`*`)`

Description Inserts the unsigned integer *`value`* into the MIME field located at *`offset`* within the marshal buffer *`bufp`*. The *`idx`* parameter specifies where the inserted value should be put with respect to the other values already in the MIME field. If *`idx`* is `0` then the value will be prepended to the list of values in the field. Increasing values of *`idx`* will place the value further down the list of values. If *`idx`* is `-1` then the value will be appended to the list of values. Normal usage is to specify `-1` for *`idx`* so that the value will be appended to the list of values. All values are stored as strings within the MIME field. `INKMimeFieldValueInsertUint` simply formats the unsigned integer into a string and then calls `INKMimeFieldValueInsert`.

Release the returned `INKMLoc` with a call to `INKHandleMLocRelease`.

First release Traffic Server 3.0

INKMimeFieldValuesClear

Clears all values in a MIME field.

Prototype void **`INKMimeFieldValuesClear`** `(INKMBuffer` *`bufp`*`, INKMLoc` *`offset`*`)`

Description Removes and destroys all of the values within the MIME field located at *`offset`* within the marshal buffer *`bufp`*.

First release Traffic Server 3.0

INKMimeFieldValuesCount

Counts the values in a MIME field.

Prototype `int` **`INKMimeFieldValuesCount`** `(INKMBuffer` *`bufp`*`, INKMLoc` *`offset`*`)`

Description	Returns a count of the number of values in the MIME field located at *offset* within the marshal buffer *bufp*.
First release	Traffic Server 3.0

INKMimeFieldValueSet

Sets a value in a MIME field.

Prototype	`void` **`INKMimeFieldValueSet`** `(INKMBuffer` *bufp*`, INKMLoc` *offset*`, int` *idx*`, const char *`*value*`, int` *length*`)`
Description	Sets a value in the MIME field located at *offset* within the marshal buffer *bufp* to the string *value*. If *length* is `-1` then it is assumed that *value* is null-terminated. Otherwise, the length of the string *value* is taken to be *length*. The string is copied to within *bufp*, so it is okay to modify or delete *value* after calling `INKMimeFieldValueSet`. The *idx* parameter specifies which value in the field to change. If *idx* is not between `0` and `INKMimeFieldValuesCount` (*bufp*, *offset*) `- 1` then no operation will be performed.
First release	Traffic Server 3.0

INKMimeFieldValueSetDate

Sets a date value in a MIME field.

Prototype	`void` **`INKMimeFieldValueSetDate`** `(INKMBuffer` *bufp*`,INKMLoc` *offset*`, int` *idx*`, time_t` *value*`)`
Description	Sets a value in the MIME field located at *offset* within the marshal buffer *bufp* to the data *value*. The *idx* parameter specifies which value in the field to change. If *idx* is not between `0` and `INKMimeFieldValuesCount` (`bufp`, `offset`) `- 1` then no operation will be performed. All values are stored as strings within the MIME field. `INKMimeFieldValueSetDate` simply formats the date into a string and then calls `INKMimeFieldValueSet`.
First release	Traffic Server 3.0

INKMimeFieldValueSetInt

Sets an integer value in a MIME field.

Prototype	`void` **`INKMimeFieldValueSetInt`** `(INKMBuffer` *bufp*`,INKMLoc` *offset*`, int` *idx*`, int` *value*`)`
Description	Sets a value in the MIME field located at *offset* within the marshal buffer *bufp* to the integer *value*. The *idx* parameter specifies which value in the field to change. If *idx* is not between `0` and `INKMimeFieldValuesCount` (*bufp*, *offset*) `- 1` then no operation will be performed. All values are stored as strings within the MIME field. `INKMimeFieldValueSetInt` simply formats the integer into a string and then calls `INKMimeFieldValueSet`.
First release	Traffic Server 3.0

INKMimeFieldValueSetUint

Sets an unsigned integer value in a MIME field.

Prototype `void` **`INKMimeFieldValueSetUint`** `(INKMBuffer` *`bufp`*`,INKMLoc` *`offset`*`, int` *`idx`*`, unsigned int` *`value`*`)`

Description Sets a value in the MIME field located at *`offset`* within the marshal buffer *`bufp`* to the unsigned integer *`value`*. The *`idx`* parameter specifies which value in the field to change. If *`idx`* is not between `0` and `INKMimeFieldValuesCount` (`bufp`, `offset`) `- 1` then no operation will be performed. All values are stored as strings within the MIME field. `INKMimeFieldValueSetUint` simply formats the unsigned integer into a string and then calls `INKMimeFieldValueSet`.

First release Traffic Server 3.0

INKMimeHdrFieldValueGet

Gets a specified field value from a MIME header.

Prototype `const char*` **`INKMimeHdrFieldValueGet`** `(INKMBuffer` *`bufp`*`, INKMLoc` *`hdr_loc`*`, INKMLoc` *`field`*`, int` *`idx`*`, int *`*`value_len_ptr`*`)`

Description Retrieves a string value from within the MIME field located at *`field`* within the marshal buffer *`bufp`*. The *`idx`* parameter specifies which field to retrieve. The fields are numbered from `0` to `INKMimeHdrFieldValuesCount` (*`bufp`*, *`hdr, field`*) `- 1`. If *`idx`* does not lie within that range then `NULL` will be returned. The length of the returned string is placed in the *`value_len_ptr`* argument. If *`value_len_ptr`* is NULL then no attempt is made to dereference it.

This API has been deprecated by `INKMimeHdrFieldValueStringGet`.

Returns A pointer to the specified field value in the MIME header. Release with a call to `INKHandleStringRelease`.

First release Traffic Server 3.5

INKMimeHdrFieldValueGetDate

Gets date value from a MIME field.

Prototype `time_t` **`INKMimeHdrFieldValueGetDate`** `(INKMBuffer` *`bufp`*`, INKMLoc` *`hdr`*`, INKMLoc` *`field`*`, int` *`idx`*`)`

Description Retrieves a date value from within the MIME field located at *`field`* within the marshal buffer *`bufp`*. The *`idx`* parameter specifies which field to retrieve. The fields are numbered from **`0`** to `INKMimeHdrFieldValuesCount` (*`bufp, hdr, field`*) **`-1`**. If idx does not lie within that range, `INKMimeHdrFieldValueGetDate` returns (`time_t`) **`0`**. All values are stored as strings within the MIME field. `INKMimeHdrFieldValueGetDate` parses the string value to return an integer date representation.

This API has been deprecated by `INKMimeHdrFieldValueDateGet`.

Returns The date value from the specified MIME header.

First release Traffic Server 3.5

INKMimeHdrFieldValueGetInt

Gets an integer field value in a MIME field.

Prototype `int` **`INKMimeHdrFieldValueGetInt`** `(INKMBuffer` *`bufp`*`, INKMLoc` *`hdr`*`, INKMLoc` *`field`*`, int` *`idx`*`, int *`*`value_len-ptr`*`)`

Description Retrieves an integer value from within the MIME field located at *`field`* within the marshal buffer *`bufp`*. The *`idx`* parameter specifies which value within the field to retrieve. The fields are numbered from `0` to `INKMimeHdrFieldValuesCount` (*`bufp`*, *`hdr, field`*) `- 1`. If *`idx`* does not lie within that range, `INKMimeHdrFieldValueGetInt` returns `(int) 0`. All values are stored as strings within the MIME field. `INKMimeHdrFieldValueGetInt` parses the string value to return an integer.

This API has been deprecated by `INKMimeHdrFieldValueIntGet`.

Returns The interger value from the specified MIME field.

First release Traffic Server 3.5

INKMimeHdrFieldValueGetUInt

Gets unsigned integer field value in a MIME field.

Prototype `unsigned int` **`INKMimeHdrFieldValueGetUInt`** `(INKMBuffer` *`bufp`*`, INKMLoc` *`hdr`*`, INKMLoc` *`field`*`, int` *`idx`*`)`

Description Retrieves an unsigned integer value from within the MIME field located at *`field`* within the marshal buffer *`bufp`*. The *`idx`* parameter specifies which field to retrieve. The fields are numbered from `0` to `INKMimeHdrFieldValuesCount` (*`bufp`*, *`hdr, field`*) `- 1`. If *`idx`* does not lie within that range, `INKMimeHdrFieldValueGetUnit` returns `(unsigned int) 0`. All values are stored as strings within the MIME field. `INKMimeHdrFieldValueGetUInt` parses the string value to return an unsigned integer.

It is not possible to determine if `INKMimeHdrFieldValueGetUInt` is returning an unsigned int value in error. If you need to check for errors in MIME header field values, you can fetch the header as a string and examine it. Here is some sample code that fetches MIME headers from marshal buffers into strings using `INKMimeHdrFieldValueGet` instead. The context of this example is that the plugin is processing an HTTP transaction and has access to a transaction.

This API has been deprecated by `INKMimeHdrFieldValueUIntGet`.

Returns The unsigned integer value from the specified MIME field.

Example

```
static void
handle_string (INKHttpTxn txnp, INKCont contp) {
    INKMBuffer bufp;
    INKMLoc hdr_loc;
    INKMLoc field;
    int len;
    char* output_string;
    const char* value;
/* Fetch  the transaction's client request header into a marshal buffer.
*/
    if (!INKHttpTxnClientReqGet (txnp, &bufp, &hdr_loc)) {
        INKError ("couldn't retrieve client request header\n");
        goto done;
    }
    field=INKMimeHdrFieldRetrieve(bufp, hdr_loc,
                                  INK_MIME_FIELD_CONTENT_LENGTH);

    if (!field) {
        INKError ("Content-Length field not found.\n");
        INKHandleMLocRelease (bufp, INK_NULL_MLOC, hdr_loc);
        goto done;
    }
    /* Obtain the value of the content length (normally an
     * unsigned int) as a string. */
    value=INKMimeHdrFieldValueGet (bufp, hdr_loc, field, 0, &len);

    if ((!value) || (len<=0))}
        INKHandleMLocRelease (bufp, hdr_loc, field);
        INKHandleMLocRelease (bufp, INK_NULL_MLOC, hdr_loc);
        goto done;
    }
   /* Allocate the string with an extra byte for the string terminator.
*/
    output_string = (char*) INKmalloc(len + 1);

    /* Copy the value. */
    strncpy (output_string, value, len);

    /* Terminate the string */
    output_string[len] = '\0';
/* Now that you have the MIME fields as a string, you can do
       whatever you want to do with it, for example, print it, or
       make sure it's an unsigned integer: either by using the
       atol C function or by scanning each ASCII character.  */
    INKDebug("my-plugin", "%s", output_string);

    /* Release handles and allocated memory. */
    INKHandleStringRelease (bufp, field, value);
    INKfree (output_string);
```

First release Traffic Server 3.5

INKMimeHdrFieldValueInsert

Inserts a value into a specified location within a MIME field.

Prototype
```
INKMLoc INKMimeHdrFieldValueInsert (INKMBuffer bufp, INKMLoc hdr,
    INKMLoc field, const char *value, int length, int idx)
```

Description Inserts the string *value* into the MIME field located at *field* within the marshal buffer *bufp*. If *length* is `-1` then `INKMimeHdrFieldValueInsert` assumes that *value* is null-terminated. Otherwise, the length of the string *value* is taken to be *length*. `INKMimeHdrFieldValueInsert` copies the string to within *bufp*, so it is okay to modify or delete *value* after calling `INKMimeHdrFieldValueSet`. The *idx* parameter specifies where the inserted value should be put with respect to the other values already in the MIME field. If *idx* is `0` then `INKMimeHdrFieldValueInsert` prepends the value to the list of values in the field. Increasing values of *idx* place the value further down the list of values. If *idx* is `-1`, `INKMimeHdrFieldValueInsert` appends the value to the list of values. Normal usage is to specify `-1` for *idx* so that the value is appended to the list of values.

This API has been deprecated by `INKMimeHdrFieldValueStringInsert`.

First release Traffic Server 3.5

INKMimeHdrFieldValueInsertDate

Inserts a date value into a MIME field.

Prototype
```
INKMLoc INKMimeHdrFieldValueInsertDate (INKMBuffer bufp, INKMLoc hdr,
    INKMLoc field, time_t value, int idx)
```

Description Inserts the data *value* into the MIME field located at *field* within the marshal buffer *bufp*. The *idx* parameter specifies where the inserted value should be put with respect to teh other values already in the MIME field. If *idx* is **`0`** then the value is prepended to the list of values in the field. Increasing values of *idx* places the value further down the list of values. If *idx* is **`-1`** then the value is appended to the list of values. Normal usage is to specify **`-1`** for *idx* so that the value is appended to the list of values. All values are stored as strings within the MIME field. `INKMimeHdrFieldValueInsertDate` simply formats the date into a string and then calls `INKMimeHdrFieldValueInsert`.

Note: do not use the return value (INKMLoc) of this function. Future versions will be changed to void.

This API has been deprecated by `INKMimeHdrFieldValueDateInsert`.

First release Traffic Server 3.5

INKMimeHdrFieldValueInsertInt

Inserts an integer value into a MIME field.

Prototype `INKMLoc` **`INKMimeHdrFieldValueInsertInt`** `(INKMBuffer` *`bufp`*`, INKMLoc` *`hdr`*`, INKMLoc` *`field`*`, int` *`value`*`, int` *`idx`*`)`

Description Inserts the integer *`value`* into the MIME field located at *`field`* within the marshal buffer *`bufp`*. The *`idx`* parameter specifies where the inserted value should be put with respect to the other values already in the MIME field. If *`idx`* is `0` then the value is prepended to the list of values in the field. Increasing values of *`idx`* places the value further down the list of values. If *`idx`* is `-1` then the value is appended to the list of values. Normal usage is to specify `-1` for *`idx`* so that the value is appended to the list of values. All values are stored as strings within the MIME field. `INKMimeHdrFieldValueInsertInt` simply formats the integer into a string and then calls `INKMimeHdrFieldValueInsert`.

This API has been deprecated by `INKMimeHdrFieldValueIntInsert`.

First release Traffic Server 3.5

INKMimeHdrFieldValueInsertUInt

Inserts an unsigned integer value into a MIME field.

Prototype `INKMLoc` **`INKMimeHdrFieldValueInsertUInt`** `(INKMBuffer` *`bufp`*`, INKMLoc` *`hdr`*`, INKMLoc` *`field`*`, unsigned int` *`value`*`, int` *`idx`*`)`

Description Inserts the unsigned integer *`value`* into the MIME field located at *`field`* within the marshal buffer *`bufp`*. The *`idx`* parameter specifies where the inserted value should be put with respect to the other values already in the MIME field. If *`idx`* is `0` then the value will be prepended to the list of values in the field. Increasing values of *`idx`* will place the value further down the list of values. If *`idx`* is `-1` then the value will be appended to the list of values. Normal usage is to specify `-1` for *`idx`* so that the value will be appended to the list of values. All values are stored as strings within the MIME field. `INKMimeHdrFieldValueInsertUInt` simply formats the unsigned integer into a string and then calls `INKMimeHdrFieldValueInsert`.

This API has been deprecated by `INKMimeHdrFieldValueUIntInsert`.

First release Traffic Server 3.5

INKMimeHdrFieldValueSet

Sets a value in a MIME field.

Prototype `void` **`INKMimeHdrFieldValueSet`** `(INKMBuffer` *`bufp`*`, INKMLoc` *`hdr`*`, INKMLoc` *`field`*`, int` *`idx`*`, const char *`*`value`*`, int` *`length`*`)`

Description Sets a value in the MIME field located at *`field`* within the marshal buffer *`bufp`* to the string *`value`*. If *`length`* is `-1` then it is assumed that *`value`* is null-terminated. Otherwise, the length of the string *`value`* is taken to be *`length`*. The string is copied to within *`bufp`*, so it is okay to modify or delete *`value`* after calling `INKMimeHdrFieldValueSet`. The *`idx`* parameter specifies which value in the field to change. If *`idx`* is not between `0` and `INKMimeHdrFieldValuesCount` (*`bufp`*, *`hdr`*, *`field`*) `- 1` then no operation will be performed.

This API has been deprecated by `INKMimeHdrFieldValueStringSet`.

First release Traffic Server 3.5

INKMimeHdrFieldValueSetDate

Sets a date value in a MIME field.

Prototype	`void` **`INKMimeHdrFieldValueSetDate`** `(INKMBuffer` *`bufp`*`, INKMLoc` *`hdr`*`, INKMLoc` *`field`*`, int` *`idx`*`, time_t` *`value`*`)`
Description	Sets a value in the MIME field located at *`field`* within the marshal buffer *`bufp`* to the date *`value`*. The *`idx`* parameter specifies which value in the field to change. If the *`idx`* is not between **`0`** and `INKMimeHdrFieldValuesCount` (*`bufp, hdr, field`*) **`-1`** then no operation will be performed. All values are stored as strings within the MIME field. `INKMimeHdrFieldValueSetDate` simply formats the date into a string and then calls `INKMimeHdrFieldValueSet`. This API has been deprecated by .
First release	Traffic Server 3.5

INKMimeHdrFieldValueSetInt

Sets an integer value within a MIME field.

Prototype	`void` **`INKMimeHdrFieldValueSetInt`** `(INKMBuffer` *`bufp`*`, INKMLoc` *`hdr`*`, INKMLoc` *`field`*`, int` *`idx`*`, int` *`value`*`)`
Description	Sets a value in the MIME field located at *`field`* within the marshal buffer *`bufp`* to the integer *`value`*. The *`idx`* parameter specifies which value in the field to change. If *`idx`* is not between `0` and `INKMimeHdrFieldValuesCount` (*`bufp`*, *`hdr, field`*) `- 1` then no operation will be performed. All values are stored as strings within the MIME field. `INKMimeHdrFieldValueSetInt` simply formats the integer into a string and then calls `INKMimeHdrFieldValueSet`. This API has been deprecated by `INKMimeHdrFieldValueIntSet`.
First release	Traffic Server 3.5

INKMimeHdrFieldValueSetUInt

Sets a value in a MIME field to a specified unsigned integer.

Prototype	`void` **`INKMimeHdrFieldValueSetUInt`** `(INKMBuffer` *`bufp`*`, INKMLoc` *`hdr`*`, INKMLoc` *`field`*`, int` *`idx`*`, unsigned int` *`value`*`)`
Description	Sets a value in the MIME field located at *`field`* within the marshal buffer *`bufp`* to the unsigned integer *`value`*. The *`idx`* parameter specifies which value in the field to change. If *`idx`* is not between `0` and `INKMimeHdrFieldValuesCount` (bufp, *`hdr, field`*) `- 1` then no operation will be performed. All values are stored as strings within the MIME field. `INKMimeHdrFieldValueSetUInt` simply formats the unsigned integer into a string and then calls `INKMimeHdrFieldValueSet`. This API has been deprecated by `INKMimeHdrFieldValueUIntSet`.
First release	Traffic Server 3.5

INKMimeHdrFieldDelete

Destroys a MIME header field.

Prototype	`void` **`INKMimeHdrFieldDelete`** `(INKMBuffer` *`bufp`*`, INKMLoc` *`hdr_loc`*`, INKMLoc` *`field`*`)`
Description	Deletes the MIME field located at *`field`* within the MIME header located at *`hdr_loc`* in the marshal buffer *`bufp`*. Make sure you release the `INKMLoc` handle *`field`* with a call to `INKHandleMLocRelease`. This API has been deprecated by `INKMimeHdrFieldDestroy`.
First release	Traffic Server 3.0

INKMimeHdrFieldInsert

Appends a field in a MIME header.

Prototype	`void` **`INKMimeHdrFieldInsert`** `(INKMBuffer` *`bufp`*`, INKMLoc` *`hdr_loc`*`, INKMLoc` *`field,`* `int` *`idx`*`)`
Description	Appends the MIME field located at *`field`* within the marshal buffer *`bufp`* into the MIME header located at *`hdr_loc`* within the marshal buffer *`bufp`*. The *`idx`* parameter specifies where the inserted field should be put with respect to the other fields already in the MIME header. This API has been deprecated by `INKMimeHdrFieldAppend`
First release	Traffic Server 3.0

INKMimeHdrFieldRetrieve

Retrieves a MIME header field.

Prototype	`INKMLoc` **`INKMimeHdrFieldRetrieve`** `(INKMBuffer` *`bufp`*`, INKMLoc` *`hdr_loc`*`, const char*` *`*retrieved_str`*`)`
Description	Retrieves a MIME field from within the MIME header located at *`hdr_loc`* within the marshal buffer *`bufp`*. The *`retrieved_str`* parameter specifies which field to retrieve. For each MIME field in the MIME header, a pointer comparison is done between the field name and *`retrieved_str`*. This is a much quicker retrieval function than `INKMimeHdrFieldFind` since it obviates the need for a string comparision. However, *`retrieved_str`* must be one of the pre-defined field names listed above of the form `INK_MIME_FIELD_XXX` for the call to succeed. If the requested field cannot be found then `0` is returned. Release with a call to `INKHandleMLocRelease`. This API has been deprecated by `INKMimeHdrFieldFind`.
First release	Traffic Server 3.0

Other Deprecated Functions

Statistic Functions

INKStatFloatRead

Obtains the value of a float stat.

Prototype	`float` **`INKStatFloat`**`(INKStat` *`the_stat`* `)` This API has been deprecated by `INKStatFloatGet`.
First release	Traffic Server 3.5

INKStatIntRead

Obtains the value of an integer stat.

Prototype	`INK64` **`INKStatIntRead`**`(INKStat` *`the_stat`* `)` This API has been deprecated by `INKStatIntGet`.
First release	Traffic Server 3.5

IO Buffer Interface

INKIOBufferAppend

Appends to an IO buffer.

Prototype	`INKReturnCode` **`INKIOBufferAppend`** `(INKIOBuffer` *`bufp`*`,` `INKIOBufferBlock` *`blockp`*`)`
Description	Appends a block to the IO buffer *`bufp`*. The data in the appended block is made available for reading.
Returns	`INK_SUCCESS` if the block was successfully appended to the specified IO buffer. `INK_ERROR` if an error occurred.
First release	Traffic Server 3.0

INKIOBufferBlockCreate

Creates an IO buffer block.

Prototype	`INKIOBufferBlock` **`INKIOBufferBlockCreate`** `(INKIOBufferData` *`datap`*`, int` *`size`*`, int` *`offset`*`)`
Description	Creates a new IO buffer block and initializes it with the IO buffer data *`datap`*. The *`size`* parameter is the amount of data that is initially available for reading in this new buffer block. The *`offset`* parameter is the offset into *`datap`* at which will be used as the start for the block. The two common uses for `INKIOBufferBlockCreate` are to create an empty block by specifying *`size`* as `0` and to create a full block by specifying *`size`* as the total size of *`datap`*. The newly created block should be added almost immediately to an IO buffer by a call to `INKIOBufferAppend` since there is no function for destroying a buffer block other than relying on it automatically being destroyed by an IO buffer.
Returns	The newly created IO buffer block.
First release	Traffic Server 3.0

INKIOBufferDataCreate

Creates IO buffer data.

Prototype	`INKIOBufferData` **`INKIOBufferDataCreate`** `(void*` *`data`*`, int` *`size`*`, INKIOBufferDataFlags` *`flags`*`)`

Description	Creates a new IO buffer data and initialize it with *`data`*, *`size`*. The *`flags`* parameter specifies how to interpret *`data`*.
	`INK_DATA_ALLOCATE`
	The *`data`* pointer is `NULL` and the data associated with the `INKIOBufferData` should be allocated. `INKIOBufferDataCreate` rounds *`size`* to a power of 2 less than or equal to 32K.
	`INK_DATA_MALLOCED`
	The *`data`* pointer was allocated by `INKmalloc` and will be freed when the last reference to the new `INKIOBufferData` is released by a call to `INKfree`.
	`INK_DATA_CONSTANT`
	The *`data`* pointer is data that should not be freed when the last reference to the new `INKIOBufferData` is released.

Returns	A handle to the newly created IO buffer.
First release	Traffic Server 3.0

Mutex function

InkMutexTryLock

Tries to lock an `INKMutex`.

Prototype	`INKReturnCode` **`InkMutexTryLock`** `(INKMutex` *`mutex,`* `int` *`*is_mutex_lock`*`)`
Description	Tries to lock the `INKMutex` *`mutex`*. In general, use `InkMutexTryLock` to obtain a mutex. See the example below. This API has been deprecated by `INKMutexLockTry`.

Returns If the mutex was successfully locked, `1` will be returned.
If *`mutex`* is already locked then `0` will be returned.

Example

```
int handler (INKCont contp, INKEvent event, void *edata)
{
   //this continuation tries to grab a mutex
   int lock = InkMutexTryLock (mutex);
   if (!lock)
   {
   /* Schedule a retry; RETRY_TIME should be 10 ms or longer. */
      INKContSchedule (contp, RETRY_TIME);
      return INK_EVENT_IMMEDIATE;
   }

   // Now the mutex is grabbed
   do_some_job ...
   INKMutexUnlock (mutexp);
}
```

First release Traffic Server 3.0

APPENDIX C

Troubleshooting Tips

This appendix lists the following troubleshooting tips.

Unable to Compile Plugins

The process you use to compile a shared library will vary from platform to platform, so the Traffic Edge API includes makefile templates you can use to create shared libraries on all the supported Traffic Edge platforms.

Unix example

Assuming the sample program is stored in the file `hello-world.c`, you could use the following commands to building a shared library on Solaris using the GNU C compiler.

```
gcc -g -Wall -fPIC -o hello-world.o -c hello-world.c
gcc -g -Wall -shared -o hello-world.so hello-world.o
```

The first command compiles `hello-world.c` as Position Independent Code (PIC) and the second command links the single `hello-world.o` object file into the `hello-world.so` shared library.

Caution

Make sure that your plugin is **not** statically linked with system libraries.

HPUX example

Assuming the sample program is stored in the file `hello_world.c`, you could use the following commands to build a shared library on HPUX:

```
cc +z -o hello_world.o -c hello_world.c
ld -b -o hello_world.so hello_world.o
```

Compiling for Windows NT

Your PC must have the following software installed:

- Windows NT 4.0 SP4
- Microsoft Developer Studio 6.0

▼ **To compile a plugin for the Windows NT version of Traffic Edge**:

1. Open `PlugIn.dsw` with Microsoft Visual C++ (MSVC++). The `dsw` file should be included in the SDK CD. Inside VC++, the sample plugins are listed as separate projects.
2. For each of the projects that need to be built, you need to tell VC++ where it can find the Traffic Edge library: `traffic_server.lib`. This library is in your NT Traffic Edge distribution.

   You might need to update the library lookup path. Use the following procedure:

▼ **To update the library lookup path**

1. Right-mouse-click on a project.
2. Select the **Settings...** option.
3. Click the **Link** tab on the dialog box.
4. Select **Input** in the combo-box.
5. Enter the library path in the **Additional library path:** text field

   Now you can build your plugin.

Unable to Load Plugins

To load plugins, follow the steps below.

1. Make sure that your plugin source code contains an `INKPluginInit` initialization function.
2. Compile your plugin source code, creating a shared library.
3. Add an entry to the `plugin.config` file for your plugin.
4. Add the path to your plugin shared library to the `records.config` file.
5. Restart Traffic Edge.

For detailed information on each step, refer to the section *"A simple plugin"* in Chapter 1.

Using Debug Tags

Use the API `void` **`INKDebug`** `(const char *`*`tag`*`, const char *`*`format_str`*`, ...)` to add traces in your plugin, where:

- *`tag`* is the Traffic Edge parameter that enables Traffic Edge to print out *`format_str`*.
- `...` is a variable for *`format_str`*.

`INKDebug` prints out the statement `format_str` if debugging is enabled. There are two ways to enable debugging:

- On UNIX systems, run Traffic Edge with the `-Ttag` option. For example, if the tag is `my-plugin`:
  `traffic_server -T"my-plugin"`
  In this case, the debug output goes to `traffic.out`.
- On either UNIX or Windows NT systems, set the following variables in `records.config` (in the Traffic Edge `config` directory):
  `proxy.config.diags.debug.enabled INT 1`
  `proxy.config.diags.debug.tags STRING` *`debug-tag-name`*
  In this case, debug output goes to `traffic.out` on UNIX systems, and to `diags.log` on Windows NT systems.

`Example:`

`INKDebug` ("my-plugin", "Starting my-plugin at %d\n", the_time);

The statement "Starting my-plugin at <time>" appears whenever you run Traffic Edge with the my-plugin tag:

`traffic_server -T"my-plugin"`

Other useful internal debug tags

Traffic Edge provides many debug tags for internal debugging purposes. Some of the useful HTTP debug tags are:

- http_hdrs - traces all incoming and outgoing HTTP headers.
- http.* - traces all the STTP SM debug statements.
- sdk - gives some warning concerning API usage.

Using a Debugger

A debugger can set breakpoints in a plugin. Use a Traffic Edge debug build and compile the plugin with the `-g` option. A debugger can also be used to analyze a core dump. To generate core, set the size limit of the core files in the `records.config` file to `-1` as follows:

`CONFIG proxy.config.core_limit INT -1`

Debugging Tips:

- Use a Traffic Edge debug version.
- Use assertions in your plugin (`INKAssert` / `INKReleaseAssert`).

Debugging Memory Leaks

Memory leaks in a plugin can be detected using a TS MRTG graph related to memory. You can use memory dump information. Enable mem dump in `records.config` as follows:

`CONFIG proxy.config.dump_mem_info_frequency INT <value>`

This causes Traffic Edge to dump mem info in `traffic.out at <value>` intervals will be in *secs*. A zero value means disabled.

Concept Index

Constant Index

Function Index

Type Index

COPYRIGHT NOTICES

Portions of Traffic Server include third party technology used under license. One or more of the following notices may apply in connection with the license and use of Traffic Server.

tcl-7.4 license. This software is copyrighted by the Regents of the University of California, Sun Microsystems, Inc., and other parties. The following terms apply to all files associated with the software unless explicitly disclaimed in individual files.

The authors hereby grant permission to use, copy, modify, distribute, and license this software and its documentation for any purpose, provided that existing copyright notices are retained in all copies and that this notice is included verbatim in any distributions. No written agreement, license, or royalty fee is required for any of the authorized uses. Modifications to this software may be copyrighted by their authors and need not follow the licensing terms described here, provided that the new terms are clearly indicated on the first page of each file where they apply.

IN NO EVENT SHALL THE AUTHORS OR DISTRIBUTORS BE LIABLE TO ANY PARTY FOR DIRECT, INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OF THIS SOFTWARE, ITS DOCUMENTATION, OR ANY DERIVATIVES THEREOF, EVEN IF THE AUTHORS HAVE BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

THE AUTHORS AND DISTRIBUTORS SPECIFICALLY DISCLAIM ANY WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, AND NON-INFRINGEMENT. THIS SOFTWARE IS PROVIDED ON AN “AS-IS” BASIS, AND THE AUTHORS AND DISTRIBUTORS HAVE NO OBLIGATION TO PROVIDE MAINTENANCE, SUPPORT, UPDATES, ENHANCEMENTS, OR MODIFICATIONS.

RESTRICTED RIGHTS: Use, duplication or disclosure by the government is subject to the restrictions as set forth in subparagraph (c)(1)(ii) of the Rights in Technical Data and Computer Software Clause as DFARS 252.227-7013 and FAR 52.227-19.

SSLeay-0.6.6 License. Copyright © 1995-1997 Eric Young (eay@mincm.oz.au). All rights reserved.

Redistribution and use in source code and binary forms, with or without modification, are permitted provided that the following conditions are met:

1.Redistributions of source code must retain the copyright notice, this list of conditions and the following disclaimer.

2.Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

3.All advertising materials mentioning features or use of this software must display the following acknowledgement: “This product incorporates cryptographic software written by Eric Young (eay@mincom.oz.au).” The word ‘cryptographic’ can be left out if the routines from the library being used are not cryptographic related.

4.If you include any Windows specific code (or a derivative thereof) from the apps directory (application code) you must include an acknowledgement “This product includes software written by Tim Hudson (tjh@mincom.oz.au)”

THIS SOFTWARE IS PROVIDED BY ERIC YOUNG “AS IS” AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE AUTHOR OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

RSAREF (for MD5). Copyright © 1991-2, RSA Data Security, Inc. Created 1991. All rights reserved.License to copy and use this software is granted provided that it is identified as the “RSA Data Security, Inc. MD5 Message-Digest Algorithm” in all material mentioning or referencing this software or this function.

License to copy and use this software is granted provided that it is identified as the “RSA Data Security, Inc. MD5 Message-Digest Algorithm” in all material mentioning or referencing this software or this function.

License is also granted to make and use derivative works provided that such works are identified as “derived from the RSA Data Security, Inc. MD5 Message-Digest Algorithm” in all material mentioning or referencing the derived work.

RSA Data Security, Inc. makes no representations concerning either the merchantability of this software or the suitability of this software for any particular purpose. It is provided “as is” without express or implied warranty of any kind.

These notices must be retained in any copies of any part of this documentation and/or software.

Portions of Traffic Server include technology used under license from RSA Data Security, Inc.

libdb-1.85 License. Copyright © 1990, 1993, 1994 The Regents of the University of California. All rights reserved.

Redistribution and use in source code and binary forms, with or without modification, are permitted provided that the following conditions are met:

1.Redistributions of source code must retain the copyright notice, this list of conditions and the following disclaimer.

2.Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

3.All advertising materials mentioning features or use of this software must display the following acknowledgement: This product includes software developed by the University of California, Berkeley and its contributors.

4.Neither the name of the University nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE REGENTS AND CONTRIBUTORS “AS IS” AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE REGENTS OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

Gateway Daemon, Release 4 license. © The Regents of the University of Michigan all rights reserved. Gate Daemon was originated and developed through release 3.0 by Cornell University and its collaborators.

Copyright notices and other restrictions as they currently appear in the GateD source files include one or more of the following:

Copyright © 1995 The Regents of the University of Michigan. All rights reserved. Gate Daemon was originated and developed through release 3.0 by Cornell University and its collaborators.

THIS SOFTWARE IS PROVIDED “AS IS” AND WITHOUT ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, WITHOUT LIMITATION, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.

Royalty-free licenses to redistribute GateD Release 2 in whole or in part may be obtained by writing to: Gate Daemon Project, The University of Michigan, Merit, 4251 Plymouth Road, Suite C, Ann Arbor, MI 48105-2785, (313) 936-9430

GateD is based on Kirton's EGP, UC Berkeley's routing daemon (routed), and DCN's HELLO routing Protocol. Development of GateD has been supported in part by the National Science Foundation.

Please forward bug fixes, enhancements and questions to the GateD mailing list: gated-bug@gated.merit.edu.

Cornell Authors: Jeffrey C. Honig, Scott W. Brim

Authors: Robert Hagens and Dan Schuh

Royalty-free licenses to redistribute UMD OSPF are available from the University of Maryland, College Park. For details contact: Office of Technology Liaison, 4312 Knox Road, University of Maryland, College Park, Maryland 20742, (301) 405-4209, (301) 314-9871 fax

This software was written by Rob Coltun. rcoltun@ni.umd.edu

gd 1.3 graphics library.

Portions copyright 1994, 1995, 1996, 1997, 1998, by Cold Spring Harbor Laboratory. Funded under Grant P41-RR02188 by the National Institutes of Health.

Portions copyright 1996, 1997, 1998, by Boutell.Com, Inc.

GIF decompression code copyright 1990, 1991, 1993, by David Koblas (koblas@netcom.com).

Non-LZW-based GIF compression code copyright 1998, by Hutchison Avenue Software Corporation (http://www.hasc.com/, info@hasc.com).

libregx package. Copyright 1992, 1993, 1994, 1997 Henry Spencer. All rights reserved. This software is not subject to any license of the American Telephone and Telegraph Company or of the Regents of the University of California.

Permission is granted to anyone to use this software for any purpose on any computer system, and to alter it and distribute it, subject to the following restrictions:

1. The author is not responsible for the consequences or use of this software, no matter how awful, even if they arise from flaws in it.

2. The origin of this software must not be misrepresented, either by explicit claim or by omission. Since few users ever read sources, credits must appear in the documentation.

3. Altered versions must be plainly marked as such, and must not be misrepresented as being the original software. Since few users ever read sources, credits must appear in the documentation.

4. This notice may not be removed or altered.

Emanate. Licensee agrees to preserve and reproduce the copyright notices contained in the Program Source and Software in the same form and location as any legend appearing on or in the original from which copies are made.

Portions of Traffic Server include Emanate software developed by SNMP Research International, Incorporated. Copying and distribution is by permission of SNMP Research International, Incorporated, and relevant third parties.

INN. Portions of Traffic Server include software developed by Rich Salz. Copyright 1991 Rich Salz. All rights reserved. Revision: 1.4

Redistribution and use in any form are permitted provided that the following restrictions are met:

1. Source distributions must retain this entire copyright notice and comment.

2. Binary distributions must include the acknowledgement “This product includes software developed by Rich Salz.” in the documentation or other materials provided with the distribution. This must not be represented as an endorsement or promotion without specific prior written permission.

3. The origin of this software must not be misrepresented, either by explicit claim or by omission. Credits must appear in the source and documentation.

4. Altered versions must be plainly marked as such in the source and documentation and must not be misrepresented as being the original software.

THIS SOFTWARE IS PROVIDED “AS IS” AND WITHOUT ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, WITHOUT LIMITATION, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.

NetFactory, Inc. Portions of Traffic Server include technology used under license from NetFactory, Inc.

IP-Filter package. Portions of Traffic Server include technology used under license from Darren Reed.