The Apache Portable Runtime (APR)

The Apache Portable Runtime (APR) and Utilities (APR-UTILS or APU) are a pair of libraries used by the Apache httpd, but autonomously developed and maintained within the Apache group. Although many core developers are involved in both httpd (the webserver) and APR, the projects are separate. These libraries provide core functions that are not specific to webserving but are also useful in more general applications.

The best-known other application is Subversion, a revision and change control management system . Another is this author's Site Valet, a suite of software for QA and accessibility audit on the web.

This chapter discusses APR as it applies to Apache modules. It does not go into subjects such as application initialisation, which are necessary but are handled internally by the Apache core code (this is documented fairly clearly within APR itself, for developers working outside the webserver context).

1 APR

The main purpose of APR is to provide a portable, platform-independent layer for applications. Functions such as filesystem access, network programming, process and thread management, and shared memory are supported in a low-level, cross-platform library. Apache modules that use exclusively APR instead of native system functions are portable across platforms, and can expect to compile cleanly – or at worst with a trivial amount of tidying up – on all platforms supported by Apache.

Each APR module comprises an application programming interface (API) shared between all platforms, together with implementations of the functions defined in the API. The implementations are often wholly or partly platform-specific, but this is of no concern to applications.

At the core of APR is Apache's resource management (pools), on which more below. The full list of APR modules is

Name	Purpose
`apr_allocator`	used internally for memory allocation
`apr_atomic`	atomic operations
`apr_dso`	dynamic loading of code (.so/.dll)
`apr_env`	reading/setting environment variables
`apr_errno`	defines error conditions and macros
`apr_file_info`	properties of filesystem objects and paths
`apr_file_io`	filesystem I/O
`apr_fnmatch`	filesystem pattern-matching
`apr_general`	initialisation/termination; useful macros
`apr_getopt`	command arguments

Name	Purpose
apr_global_mutex	global locking routines
apr_hash	hash tables
apr_inherit	file handle inheritance helpers
apr_lib	odds and sods
apr_mmap	memory mapping
apr_network_io	network i/o (sockets)
apr_poll	poll routines
apr_pools	resource management
apr_portable	APR <--> native mapping conversion
apr_proc_mutex	process locking routines
apr_random	random numbers
apr_ring	ring data struct and macros
apr_shm	shared memory
apr_signal	signal handling
apr_strings	string operations
apr_support	internal support function
apr_tables	table and array functions
apr_thread_cond	thread conditions
apr_thread_mutex	thread mutex routines
apr_thread_proc	threads and process functions
apr_thread_rwlock	reader/writer lock routines
apr_time	time/date functions
apr_user	user and group ID services
apr_version	APR version
apr_want	standard header support

2 APR-UTIL

APR-UTIL is a second library in the APR project. The purpose of APU is to provide a small set of utilities, based on the APR and with a unified programming interface. APU doesn't have separate per-platform modules, but does adopt a similar approach to some other resources commonly used, such as databases.

A complete list of APU modules is

Name	Purpose
apr_anylock	transparent any lock flavor wrapper
apr_base64	base64 encoding
apr_buckets	buckets/bucket brigades
apr_date	date string parsing
apr_dbd	common API for SQL databases
apr_dbm	common API for DBM databases
apr_hooks	hook implementation macros
apr_ldap	LDAP
apr_ldap_init	
apr_ldap_option	
apr_ldap_url	
apr_md4	MD4 encoding
apr_md5	MD5 encoding
apr_optional	optional functions
apr_optional_hooks	optional hooks
apr_queue	thread-safe FIFO queues
apr_reslist	pooled resources
apr_rmm	relocatable managed memory
apr_sdbm	SDBM library
apr_sha1	SHA1 encoding
apr_strmatch	string pattern matching
apr_uri	URI parsing/construction
apr_uuid	
apr_xlate	charset conversion (I18N)
apr_xml	XML parsing

3 Basic Conventions

APR and APR-UTIL adopt a number of conventions throughout, that give them a homogenous API and make them easy to work with.

Reference Manual: API Documentation and Doxygen

All of APR/APU is very well documented at the code level. Every public function and datatype is documented in the header file that defines it, in doxygen-friendly format. The header files themselves, or the doxygen-generated documentation, provide a full API reference for programmers.

Namespacing

All APR/APU public interfaces are prefixed with the string "`apr_`" (data types and functions) or "`APR_`" (macros). This defines APR's 'reserved' namespace.

Within the APR namespace, most of the different APR and APU modules use secondary namespacing. This is often based on the name of the module in question: for example, all functions in module `apr_dbd` are prefixed with the string "`apr_dbd_`". Sometimes an obviously descriptive secondary namespace is used: for example, socket operations in module `apr_network_io` are prefixed "`apr_socket_`".

Declaration macros

Public functions in APR/APU are declared using macros such as `APR_DECLARE`, `APU_DECLARE`, and `APR_DECLARE_NONSTD`. For example:

```
APR_DECLARE(apr_status_t) apr_initialize(void);
```

On most platforms, this is a null declaration and expands simply to

```
apr_status_t apr_initialize(void);
```

but on platforms such as Windows with Visual C++, which require their own nonstandard keywords such as `_dllexport` to enable other modules to use a function, these macros will expand to the required keywords.

apr_status_t and return values

A convention widely adopted in APR/APU is that functions return a status value indicating success or an error code to the caller. The type is an enumeration `apr_status_t`, defined in `apr_errno.h`. Thus the usual prototype for an APR function is

```
APR_DECLARE(apr_status_t) apr_do_something(...function args...);
```

Return values should routinely be tested, and error handling (recovery or graceful failure) implemented. The return value APR_SUCCESS indicates success, so we can commonly handle errors using constructs like

```
apr_status_t rv;
...
rv = apr_do_something(... args ...);
if (rv != APR_SUCCESS) {
    /* log an error */
    return rv;
}
```

Sometimes we may do more: for example, if `do_something` was a non-blocking I/O operation and returned `APR_EAGAIN`, we will probably want to retry.

Some functions return a string value (`char*` or `const char*`), a `void*`, or `void`. These are assumed to have no failure conditions.

Conditional Compilation

By its nature, there are a number of features of APR that may not be supported on every platform. For example, FreeBSD prior to Version 5.x had no native thread implementation considered suitable for Apache, so threads were not supported in APR (unless the relevant options were set manually for compilation).

To enable applications to work with this, APR provides `APR_HAS_*` macros for such features. When an application is concerned with such a feature, it should use conditional compilation based on these macros. For example, a module performing an operation that could lead to a race condition in a multithreaded environment may want to use something like:

```
#if APR_HAS_THREADS
    rv = apr_thread_mutex_lock(mutex);
    if (rv != APR_SUCCESS) {
        /* log an error */
        goto unlock;
    }
#endif

    /* Execute critical section of code here */

#if APR_HAS_THREADS
unlock:
    apr_thread_mutex_unlock(mutex);
#endif
```

4 Resource Management: APR Pools

The APR pools are a fundamental building block at the heart of APR and Apache, and are the basis for all resource management. They serve to allocate memory, either directly (in a malloc-like manner) or indirectly (e.g. in string manipulation), and, crucially, ensure that memory is freed at the appropriate time. But they extend much further, to ensure that other resources such as files or mutexes can be allocated and will always be properly cleaned up. They can even deal with resources managed opaquely by third-party libraries.

The Problem of Resource Management

Every programmer knows that when you allocate a resource, you must ensure it is released again when you've finished with it. For example:

```
char* buf = malloc(n) ;
... check buf is non null ...
... do something with buf ...
free(buf) ;
```

or

```
FILE* f = fopen(path, "r") ;
... check f is non null ...
... read from f ....
fclose(f) ;
```

Clearly, failure to free `buf` or to close `f` is a bug, and in the context of a long-lasting program such as Apache it would have serious consequences, up to and including bringing the entire system down. So it is important to get it right.

In trivial cases this is straightforward. But in a more complex case with multiple error paths, and even the scope of a resource being uncertain at the time it is allocated, it becomes a serious problem to ensure that cleanup takes place in every execution path. So we need a better way to manage resources.

Constructor/Destructor model

One method of resource management is exemplified by the C++ concept of objects having a constructor and destructor. A method adopted by many C++ programmers is to make the destructor responsible for cleanup of all resources allocated by the object. This approach works well provided all dynamic resources are clearly made the responsibility of an object. But, as with the simple C approach, it requires a good deal of care and attention to detail - for example where resources are conditionally allocated, or shared between many different objects - and is vulnerable to programming bugs.

Garbage Clearance Model

A high-level method of resource management, typified by Lisp and Java, is garbage-clearance. This has the advantage of taking the problem right away from the programmer and transferring it to the language itself, so the danger of crippling programming errors is removed altogether. The drawback is that it is a substantial overhead even where it isn't necessary, and it deprives the programmer of useful levels of control, such as the ability to control the lifetime of a resource. It also requires that all program components - including third-party libraries - are built on the same system, which is clearly not possible in an open system written in C.

APR Pools

The APR pools provide an alternative model for resource management. Like garbage collection, they liberate the programmer from the complexities of dealing with cleanups in all possible cases. But they offer several additional advantages, including full control over the lifetime of resources, and the ability to manage heterogenous resources.
The basic concept is that whenever you allocate a resource that requires cleanup, you register it with a pool. The pool then takes responsibility for the cleanup, which will happen when the pool itself is cleaned. That means that the problem is reduced to one of allocating and cleaning up a single resource: the pool itself. And since the Apache pools are managed by the server itself, the complexity is removed from applications

programming. All the programmer has to do is select the appropriate pool for the required lifetime of a resource.

Basic Memory Management

The most basic usage of pools is for memory management. Instead of

```
mytype* myvar = malloc(sizeof(mytype)) ;
/* make sure it gets freed later in every possible execution path */
```

we use

```
mytype* myvar = apr_palloc(pool, sizeof(mytype)) ;
```

and the pool automatically takes responsibility for freeing it, regardless of what may happen in the meantime.

This takes many forms in APR and Apache, where memory is allocated within another function. Examples are string-manipulation functions and logging, where we gain the immediate benefit of being able to use constructs like the APR version of `sprintf()` without having to know the size of a string in advance:

```
char* result = apr_psprintf(pool, fmt, ...) ;
```

APR also provides higher-level abstractions of pool memory, for example in the buckets used to pass data down the filter chain.

Generalised Memory Management

APR provides inbuilt functions for managing memory, and a few other basic resources such as files, sockets, and mutexes. But there is no requirement to use these. An alternative is to use native allocation functions, and explicitly register a cleanup with the pool:

```
mytype* myvar = malloc(sizeof(mytype)) ;
apr_pool_cleanup_register(pool, myvar, free,
        apr_pool_cleanup_null) ;
```

or

```
FILE* f = fopen(filename, "r") ;
apr_pool_cleanup_register(pool, f, fclose, apr_pool_cleanup_null) ;
```

will delegate responsibility for cleanup to the pool, so that no further action from the programmer is required. However, native functions may be less portable than APR equivalents from `apr_pools` and `apr_file_io` respectively.

This method generalises to resources opaque to Apache and APR. For example, to open a database connection and ensure it is closed after use:

```
MYSQL* sql = NULL ;
sql = mysql_init(sql) ;
if ( sql == NULL ) { log error and return failure ; }
apr_pool_cleanup_register(pool, sql, mysql_close,
        apr_pool_cleanup_null) ;

sql = mysql_real_connect(sql, host, user, pass,
        dbname, port, sock, 0) ;
if ( sql == NULL ) { log error and return failure ; }
```

Note that `apr_dbd` provides an altogether better method for managing database connections.

As a second example, consider XML processing:

```
xmlDocPtr doc = xmlReadFile(filename)
apr_pool_cleanup_register(pool, doc, xmlFreeDoc,
        apr_pool_cleanup_null) ;

/* now do things with doc, that may allocate further memory
   managed by the XML library but will be cleaned by xmlFreeDoc
 */
```

Integrating C++ destructor-cleanup code provides yet another example. Suppose we have:

```
class myclass {
public:
    virtual ~myclass() { do cleanup ; }
    // ....
} ;
```

We define a C wrapper:

```
void myclassCleanup(void* ptr) { delete (myclass*)ptr ; }
```

and register it with the pool when we allocate myclass:

```
myclass* myobj = new myclass(...) ;
apr_pool_cleanup_register(pool, (void*)myobj, myclassCleanup,
        apr_pool_cleanup_null) ;

// now we've hooked our existing resource management from C++
// into apache and never need to delete myobj
```

Implicit and Explicit Cleanup

Now, supposing we want to free our resource explicitly before the end of the request - for example, because we're doing something memory-intensive but have objects we can free. We may want to do everything according to normal scoping rules, and just use pool-based cleanup as a fallback to deal with error paths. However, since we registered the cleanup,

it will run regardless, and could lead to a double-free and a segfault.

Another pool function, `apr_pool_cleanup_kill`, is provided to deal with this situation. When we run the explicit cleanup, we unregister the cleanup from the pool. Or we can be a little more clever about it. Here's the outline of a C++ class that manages itself based on a pool, regardless of whether it is explicitly deleted or not:

```
class poolclass {
private:
  apr_pool_t* pool ;
public:
  poolclass(apr_pool_t* p) : pool(p) {
      apr_pool_cleanup_register(pool, (void*)this,
          myclassCleanup, apr_pool_cleanup_null) ;
  }
  virtual ~poolclass() {
      apr_pool_cleanup_kill(pool, (void*)this, myclassCleanup) ;
  }
} ;
```

If you use C++ with Apache (or APR), you can derive any class from `poolclass`. Most APR functions do something equivalent to this, using register and kill whenever resources are allocated or cleaned up.

Resource Lifetime

When we allocate resources on a pool, we ensure that they get cleaned up at some point. But when? We need to ensure the cleanup happens at the right time. Neither while the resource is still in use, nor long after it is no longer required.

Apache Pools

Fortunately, Apache makes this easy for us, by providing different pools for different types of resource. These pools are associated with relevant structures of the httpd, and have the lifetime of the corresponding struct. There are four general-purpose pools always available in Apache:

- the request pool, with the lifetime of an HTTP request
- the process pool, with the lifetime of an server process
- the connection pool, with the lifetime of a TCP connection
- the configuration pool.

The first three are associated with the relevant Apache structs, and accessed as `request->pool`, `connection->pool` and `process->pool`, respectively. The fourth, `process->pconf`, is also associated with the process, but differs from the process pool because it is cleared whenever Apache re-reads its configuration.

The process pool is suitable for long-lived resources, such as those that are initialised at server startup, or those cached for re-use over multiple requests. The request pool is suitable for transient resources used to process a single request.

A third general-purpose pool is the connection pool, which has the lifetime of a connection, being one or more Request. This is useful for transient resources that cannot be associated with a request: most notably in a connection-level filter, where the `request_rec` structure is undefined.

As well as these standard pools, special-purpose pools are created for other purposes including configuration and logging, or may be created privately by modules for their own use.

Using Pools in Apache: Processing a Request

All request-processing hooks take the form

```
int my_func(request_rec* r) {
  /* implement the request processing hook here */
}
```

This puts the request pool `r->pool` at our disposal. As discussed above, the request pool is appropriate for the vast majority of operations involved in processing a request. That's what we pass to Apache and APR functions that need a pool argument, as well as our own.

The process pool is available as `r->server->process->pool` for operations that need to allocate long-lived resources; for example, caching a resource that should be computed once and subsequently re-used in other requests.

The connection pool is `r->connection->pool`.

Using Pools in Apache: Initialisation and Configuration

The internals of Apache's initialisation are complex. But as far as modules are concerned, it can normally be treated as simple: one just sets up a configuration, and everything is permanent. Apache makes that easy: most of the relevant hooks have prototypes that pass the relevant pool as their first argument:

Configuration handlers

```
static const char* my_cfg(cmd_parms* cmd, void* cfg, /* args */ )
```

Use the configuration pool, `cmd->pool`, to give a configuration the lifetime of the directive.

Pre- and Post-config

These hooks are unusual in having several pools passed:

```
static int my_pre_config(apr_pool_t* pool,
        apr_pool_t* plog,apr_pool_t* ptemp)
```

For most purposes, just use the first pool, but if a function uses pools for temporary resources within itself, use `ptemp`.

Child init

```
static void my_child_init(apr_pool_t* pool, server_rec* s).
```

Again, the pool is the first argument.

Monitor

```
static int my_monitor(apr_pool_t* pool)
```

The monitor is a special case: it is running in the parent process and not tied to any time-limited structure. So resources allocated in a monitor function should be explicitly freed. If necessary, a monitor may create and free its own subpool. Few applications will need to use the monitor hook.

Using Pools in Apache: Other Cases

Most Apache modules involve the initialisation and request-processing we have discussed. But there are two other cases to deal with:

Connection Functions

The `pre_connection` and `process_connection` connection-level hooks pass a `conn_rec` as their first argument, and are directly analogous to request functions as far as pool resources are concerned. The `create_connection` connection-initialisation hook passes the pool as its first argument: any module implementing it takes responsibility for setting up the connection.

Filter Functions

Filter functions receive an `ap_filter_t` as their first argument. This ambiguously contains both a `request_rec` and a `conn_rec` as members, regardless of whether it is a request-level or a connection-level filter. Content filters should normally use the request pool. Connection-level filters will get a junk pointer in `f->r` and must use the connection pool. This can be a trap for the unwary.

5 Basics

APR provides a direct alternative to functions which are familiar and almost certain to be available on your system without any need for APR. Nevertheless, there are good reasons to use the APR versions of these functions:

- APR functions are platform-independent, and give better portability.
- APR functions get the benefit of APR's pool-based resource management for free.

We won't go into detail here: for more information, see the excellent documentation in the header files.

Strings and Formats

The **`apr_strings`** module provides APR implementations of

- common string functions: comparisons, substring matches, copying, concatenation
- stdio-like functions: sprintf and family, including vformatters
- parsing, including thread-safe strtok
- conversion to and from other data types (atoi, etc)

There is no regular expression support in APR (although there is in Apache), but the **`apr_strmatch`** module provides fast string matching that deals with the issues of case-insensitive (as well as case-sensitive) search, and non-null-terminated strings.

Internationalisation

The **`apr_xlate`** module provides conversion between different character sets.

At the time of writing, `apr_xlate` on the Windows platform relies on a third APR library, `apr_iconv`, because Windows lacks (or lacked) native internationalisation support. This dependency is expected to be removed in future.

Time and Date

The **`apr_time`** module provides a microsecond timer and clock, together with conversion to/from `<time.h>` types, timezones, time arithmetic, sleep, and formatting functions (ctime and RFC822 time as used in HTTP and other network protocols).

The **`apr_date`** module provides additional functions for parsing commonly-used time and date formats.

Data Structs

- **`apr_hash`** provides hash tables.
- **`apr_table`** provides tables and arrays.
- **`apr_queue`** provides FIFO queues.
- **`apr_ring`** provides a ring struct, which is also the basis for APR bucket brigades.

Filesystem

- **`apr_file_io`** provides standard file operations: open/close, stdio-style read/write operations, locking, create/delete/copy/rename/chmod. It supports ordinary files, tempfiles, directories, and pipes.
- **`apr_file_info`** provides filesystem info (stat), directory manipulation functions (open, close, read, etc), file path manipulation and relative path resolution.
- **`apr_fnmatch`** provides pattern-matching for the filesystem, to support wildcard operations.

Network

- `apr_network_io` is a socket layer supporting IPv4 and IPv6, and TCP, UDP and SCTP protocols. It supports a number of features subject to underlying operating system support, and will emulate them where not available. These include sendfile, accept filters, and multicast.
- **`apr_poll`** provides functions for polling a socket (or other descriptor).

System Users and Groups

- **`apr_user`** provides a cross-platform implementation of system users and groups.

Encoding and Crypto

APR does not provide a cryptographic library, and Apache's `mod_ssl` relies on the external OpenSSL package for implementation of transport-level security. However, it does support a number of data encoding and hashing techniques in its **`apr_base64`**, **`apr_md4`**, **`apr_md5`** and **`apr_sha1`** modules.

URI handling

- **`apr_uri`** defines a struct for URI/URLs, and provides parsing and unparsing functions

6 Databases in APR/Apache

Readers of a certain age will recollect a time in the 1980s, when every application for the PC came bundled with hundreds of different printer drivers on ever-growing piles of floppy discs. Eventually, the operating system implemented the sensible solution: a unified printing API, so that each printer had a single driver, and each application a single print function that works with any driver.

The history of database support in Apache echoes this. At first, Apache had no database support, so every module needing it had to implement it. Apache 1.3 had separate virtually identical modules for authentication with ndbm and Berkeley DB, and a whole slew of different (third-party) authentication modules for popular SQL databases such as MySQL. Similarly, every scripting language – such as Perl, PHP and Python – has its own database management.

Then in time for Apache 2.0, the `apr_dbm` module provided a unified interface for the DBM (simple key/value lookup) class of databases. Most recently, another module `apr_dbd` provides an analogous API for SQL databases. Just as with the printer drivers, the APR database classes remove the need for duplication, and are the preferred means of database support for new applications in APR and Apache.

DBMs and apr_dbm

DBMs have been with us since the early days of computing, when the need for fast keyed lookups was recognised. The original DBM is a Unix-based library and file format for

fast, highly scalable keyed access to data. It was followed in order by NDBM ("new DBM"), GDBM ("GNU DBM") and the Berkeley DB. This last is by far the most advanced, and the only DBM under active development today, but all of these, from NDBM onwards, provide the same core functionality used by most programs, including Apache.

Although NDBM is now old – like the city named NewTown ("Neapolis") by the Greeks in about 600 BC and still called Naples today – it is still the baseline DBM, and is the one used by early Apache modules such as the Apache 1.x versions of `mod_auth_dbm` and `mod_rewrite`. Both GDBM and Berkeley DB provide NDBM emulations, and Linux distributions ship with one or other of those emulations in place of the "real" NDBM, which is excluded for licensing reasons. Unfortunately, the various file formats are totally incompatible, and there are subtle differences in behaviour concerning database locking. This led to a steady stream of Linux users having problems with DBMs in Apache 1.x.

Apache 2 replaces direct access to a DBM with a unified wrapper layer, `apr_dbm`. There can be one or more underlying databases: this will be determined at build time, either as a configuration option, or by default detected automatically by the build scripts. The database to be used by an application may be passed as a parameter whenever a DBM is opened, so it is under direct programmer control (or administrator control, if the database is configurable) and can be trivially switched if that is ever necessary. Alternatively, for cases like authentication that are known to work well with any DBM, it can use a system default. Apache only has to support a single DBM interface, so for example a single DBM authentication module serves regardless of the underlying DBM used.

The `apr_dbm` layer, which is similar to the DBM APIs, is documented in `apr_dbm.h`. When programming with it, one should not assume any locking, although update operations are in fact safe if the DBM is GDBM or the original NDBM. Using a mutex for critical updates makes it safe in all cases.

The DBM functions supported in APR are basically the same as those common to all the DBMs, an API essentially equivalent to NDBM, GDBM and early versions of Berkeley DB. Advanced capabilities of recent Berkeley DB, such as transactions, are not supported, so applications requiring them have to access DB directly.

Example

The function `fetch_dbm_value` in `mod_authn_dbm` looks up a value in a DBM database.

```
static apr_status_t fetch_dbm_value(const char *dbmtype,
                                    const char *dbmfile,
                                     const char *user, char **value,
                                     apr_pool_t *pool)
{
    apr_dbm_t *f;
    apr_datum_t key, val;
    apr_status_t rv;

    rv = apr_dbm_open_ex(&f, dbmtype, dbmfile, APR_DBM_READONLY,
                         APR_OS_DEFAULT, pool);
```

```
    if (rv != APR_SUCCESS) {
        return rv;
    }

    key.dptr = (char*)user;
#ifndef NETSCAPE_DBM_COMPAT
    key.dsize = strlen(key.dptr);
#else
    key.dsize = strlen(key.dptr) + 1;
#endif

    *value = NULL;

    if (apr_dbm_fetch(f, key, &val) == APR_SUCCESS && val.dptr) {
        *value = apr_pstrmemdup(pool, val.dptr, val.dsize);
    }

    apr_dbm_close(f);

    return rv;
}
```

SQL Databases and apr_dbd

Note: `apr_dbd` is not available in APR0.x, and therefore Apache 2.0. It requires APR 1.2 or higher, or current CVS.

SQL is the standard for non-trivial database applications, and there are many databases regularly used with Apache in web applications. The most popular is the lightweight open-source MySQL, but overall it is one among many.

SQL databases are altogether bigger and more complex than DBMs, and are not in general interchangeable, except where applications are explicitly designed to be portable (or in a limited range of simple tasks). Nevertheless, a unified API for SQL applications brings benefits analogous to the printer drivers.

The `apr_dbd` module is a unified API for using SQL databases in Apache and other APR applications. The concept is similar to Perl's DBI/DBD framework, or the C libdbi, but `apr_dbd` differs from these in that APR pools are used for resource management, so it is much easier to work with in APR applications.

`apr_dbd` is also unusual within APR in its modular approach. Whereas the `apr_dbd` API is compiled into `libaprutil`, the drivers for individual databases are (by default) dynamically loaded at runtime. This means that when you install a new database package, you can install an APR driver for it without having to recompile the whole of APR or APR-UTIL.

At the time of writing, `apr_dbd` supports MySQL, PostgreSQL and SQLite databases. It is likely that drivers for Oracle and other databases will be contributed in due course.

The MySQL driver

MySQL is a special case. Because it is licensed under the GNU General Public License (GPL), a driver for it must also be distributed under the GPL (or not at all). This is incompatible with Apache licensing policy, because it would impose additional restrictions on Apache users.

The author has dealt with this issue by making a MySQL driver available separately, from http://apache.webthing.com/, and licensing it under the GPL. Users requiring this driver should download it into the `apr_dbd` directory or folder and build it there. If MySQL is installed in a standard location, it should then be automatically detected and built by the standard APR-UTIL configuration process.

Usage

Apache modules should normally use `apr_dbd` through the provider module `mod_dbd`, which is discussed in detail in Chapter 7.

7 Advanced Topics

Resource Pooling

A database is a fundamental component of many web applications. But connecting to it is an overhead that affects traditional application architectures such as CGI and the environment commonly known as LAMP (Linux, Apache, MySQL, [Perl|PHP|Python]). Using `apr_reslist` (APR's resource pooling module) with Apache 2's threaded MPMs, we can achieve significant improvements in performance and scalability in applications using 'expensive' resources such as databases.

A More Efficient LAMP

Many web-based applications generate dynamic content in whole or in part from a back-end server. Where the backend is designed as connection-oriented, there is a mismatch with the request-oriented HTTP protocol. This can easily lead to inefficiency in applications. The most common case is that of an SQL backend, where there is always an overhead to creating a connection and logging in. LAMP exemplifies this.. When the connection is TCP/IP over a network, that is an additional overhead.

Simple CGI

A CGI script services a single request. So the baseline for CGI to access a database is to open a connection, run any necessary queries, close the connection, and return content to the client.

This is fine for a low-traffic site, but grows inefficient as the hit rate rises above a few tens per minute. So as traffic rises, an alternative model is required. For CGI, we can use an alternative implementation such as FastCGI. But the most widely used architecture is LAMP.

Classic LAMP

The classic solution to this, as provided for many years by application development environments such as mod_perl and PHP, is for the Apache server process to hold a database connection open, saving the overhead of opening and closing a connection for every request. With Apache 1.x, this is essentially the best you can do, and is the usual way of working.

However, this solution has its own problems. Although it substantially reduces the per-hit overhead, it introduces another: holding a large number of backend connections open. This in itself puts a load both on the webserver itself and the backend, and limits the number of users that can be concurrently serviced. This doesn't just affect database-driven traffic: requests for static web pages also have to be served by an Apache process that is keeping an open connection to the backend.

Taking advantage of Apache 2

With Apache 2 and threaded MPMs, a wider range of altogether more efficient and scalable options present themselves. Starting from what we already have, we can list our options:

- classic CGI: one connection per request
- classic LAMP: one persistent connection per thread
- alternative LAMP: one persistent connection per process, with a mechanism for a thread to take and lock it
- connection pooling: more than one connection per process, but fewer than one per thread, with a mechanism for a thread to take and lock a connection from the pool
- dynamic connection pooling: a variable-size connection pool, that will grow or shrink according to actual database traffic levels.

“Alternative LAMP” dispenses with the LAMP overhead at the cost of preventing parallel accesses to the backend. It may be an efficient solution in some cases, but clearly presents its own problems with servicing concurrent requests.

The fourth and fifth present an optimal solution whose scalability is limited only by the available hardware and operating system. The number of backend connections to threads should reflect the proportion of the total traffic that requires the backend. So, in simple terms, if one in every five requests to the webserver requires the database, then a pool might have one connection per five threads. Just as Apache itself maintains a dynamic pool of threads to service incoming HTTP connections, so the optimal solution to managing backend connections is a dynamic pool whose size is driven by actual demand rather than best-guess configuration.

Implementation of Connection Pooling

Although the case for connection pooling is clear, implementation has been a gradual process. Its conception was around the time of ApacheCon in November 2003, when this author floated the idea in a "Birds of a Feather" session entitled "the module developers

wishlist". Having discussed it and found I was not alone in wanting it, I proceeded to implement, in addition to the Site Valet connection pooling module, open-source. PostgreSQL and MySQL connection pooling modules. Paul Querna implemented a similar module for connecting to a database with libdbi. Towards the end of 2004 this work matured into the Apache DBD framework, and `mod_dbd` was merged into the Apache source code as recently as May 2005.

The current implementation is `mod_dbd`, which is presented at length as our example of a service module in Chapter 7.

MultiProcess and Thread Support

Thread support, Mutexes and Atomics, Shared Memory, Signals

Miscellaneous

Optional Functions and Hooks

Buckets and Brigades