# Overview of DaemonCore in Condor

Todd Tannennbaum
tannenba@cs.wisc.edu

July 8, 2004


**Abstract**

This document provides a high-level overview of the DaemonCore framework.

## 1 Introduction

The DaemonCore framework attempts to consolidate all of the functionality that is common to all Condor daemons, as well as provide a basic operating system abstraction layer to shield the developer from differences between Unix and Win32. In its current form, it is indeed much more of a *framework* instead of a *library*, largely because it includes a main function. The following facilities are provided:

- A consistent core set of command-line parameters.
- Configuration of daemon parameters, including the ability to reconfigure remotely.
- Logging.
- Process management, independent of the underlying opereating system.
- A *non-preemptive* event driven framework that supports remote method invocations (commands), timers, signals, pipes, and more.

## 2 CEDAR

DaemonCore is built on top of the *CEDAR* library. Originally, CEDAR was written as a replacement for Sun's XDR library, back when the Condor Team grew frustrated that each platform had its own independent XDR implementation, each with its own subtle differences and bugs. CEDAR is a network communication library that evolved over time and serves the following purposes:

1. Convert a large number of types between different hosts in a reliable method. For example, if a client sends a server a double, and they're totally different architecture, CEDAR will reorder the bytes if need be, and convert it into the right-size (if a long is 4 bytes on one platform, and 8 on another CEDAR gets it right.)

CEDAR Client

```
{
  ReliSock s;
  char *data="foo";
  int result;

  s.connect(server.com,4499);

  /* Send the data */
  s.encode();
  s.code(data);
  s.end_of_message();

  /* Receive the result */
  s.decode();
  s.code(result);
  s.end_of_message();
  printf("Result = %d",result);
}
```

CEDAR Server

```
{
  ReliSock list,s;
  char *data=NULL;
  int result = 0;

  /* Setup the socket */
  list.bind(4499);
  list.listen();
  list.accept(s);

  /* Receive the data */
  s.decode();
  s.code(data);
  s.end_of_message();

  /* Compute/send result */
  if (data) {
    result = strlen(data);
    free(data);
  }
  s.encode();
  s.code(result);
  s.end_of_message();
}
```

Figure 1: **A simple CEDAR example.** *The client sends a string, and the server responds with the string's length.*

2. CEDAR is also our platform-independent socket and communications library that provides a C++ abstraction to Berkeley sockets. CEDAR has two types of Sockets: The `ReliSock`, which gives you a TCP-like stream, and the `SafeSock`, which gives you a UDP-based message exchange. (The SafeSock name has to do with buffering- you can just give it huge packets and it gets it right, splitting them up into fragments and reassembling it on the other side)

3. CEDAR can authenticate for you with a whole host of authentication methods - all you have to do is say "authenticate()" to a CEDAR socket, and it can use Kerberos, X509, NT LanMan, File system, Claim-to-be, and hopefully soon a password (actually, hopefully soon a PAM module, which gets us a huge new world of stuff)

4. CEDAR can encrypt everything that goes over it, either with Blowfish or 3DES. This is separate from the authentication code, so you can authenticate without encrypting, or encrypting with only a shared secret ahead of time. CEDAR can also verify the integrity of the message via MD5.

5. CEDAR can bandwidth regulate

6. CEDAR can limit itself to a range of ports

7. All blocking network calls in CEDAR can be invoked with a timeout value (in seconds).

8. CEDAR (will soon enough in the v6.7 development series) support a connection-broker approach via the Condor Project's GCB (Generic Connection Broker) technology, to allow for third parties to establish connections on a users behalf. We need this for supporting Condor installations that span across firewalls and NAT networks, where network access is not symmetric.

Practially all network communication throughout the Condor system is performed via the CEDAR library.

The CEDAR classes of primary interest, and the only ones that should be directly instantiated, are *ReliSock* and *SafeSock*. ReliSock provides communication via TCP, and SafeSock provides communication via UDP. The `encode()` method sets up the socket to send data, and the `decode()` method sets up the socket to receive – see the simple example in Figure 1. CEDAR does not encode the data type on the wire; thus, it is imperative that the

client and server "match-up" in that if the client is encoding an integer followed by a string onto a socket, the server had better be written to decode an integer followed by a string. The `end_of_message()` method, or sometimes abbreviated as `eom()`, flushes the encoded data onto the wire if the socket is currently in encode mode. If the socket is in decode mode, the `eom()` will verify that all buffered data has been consumed. If `eom()` fails in the encode direction, it means there was some error writing onto the socket. If it fails in the decode direction, it typically signals a programmer error due to a mismatch between what the client is sending -vs- what the server received. The `eom()` method should be called whenever the message is complete or whever the programmer desires to switch from encode to decode, or vice-versa.

CEDAR homoginizes data representations across different platforms. For example, on one platform an integer may be represented by 4 bytes in big-endian format, while on another platform an integer may be 8 bytes in little-endian format. When CEDAR encodes an integer, it will translate the integer from platform native format into an abstract CEDAR representation, send it on the wire, and then translate back into a native format on the receiving side.

The implementation of CEDAR can be found in `src/condor_io`, while most of the header files are located in `src/condor_includes`. The class hierarchy of CEDAR is a little bit funky, but basically the `Stream` class is the base class. The `Stream` class is mostly concerned with homoginization of data representations. The `Sock` class is derived from Stream, and defines an interface for network connection establishment. This interface is implemented with TCP in the `ReliSock` class and with UDP in the `SafeSock` class. ReliSock and SafeSock are derived from Sock, and are the only CEDAR classes which should be instantiated; however, pointers to the Stream base class are often used at points in the application where it is not important if the underlying medium is TCP or UDP (for example, in a daemon which handles the same commands over both TCP and UDP).

Throughout the code, you will see references to a `sinful string`. All this means is a string of the form `<xxx.xxx.xxx.xxx:ppp>`, i.e. an ip address plus colon plus port nunber, all enclosed in less-than greater-than characters. The sinful string is a common convention throughout CEDAR and DaemonCore for communicating a unique network endpoint. Several helpful functions for manipulating sinful strings can be found in `src/condor_util_lib/internet.c`.

The most complicated portions of CEDAR are the parts dealing with security. The ReliSock class includes an `authenticate()` method that will perform strong authentication using Kerberos, NTSSPI, GSI, or several other methods. In addition, both ReliSock and SafeSock provide the ability to encrypt data via 3DES or Blowfish algorithms, and also the perform an integrity check on the data via an MD5 checksum. The implementation for these algorithms is courtesy the OpenSSL library, which must be linked into Condor (and typically is linked via the Globus Toolkit external). More information on security in DaemonCore is included in a different document.

# 3 Understanding the Event-based Framework

DamonCore functionality is accessed via invoking methods upon the global singloton `daemonCore`.

Every DaemonCore daemon must have a global string variable called `mySubSystem` and few entrypoints. See `src/condor_dcskel` for a skeletal DaemonCore daemon as an example. The `mySubSystem` variable is used to identify the name of the service. For example, the value of `mySubSystem` for the `condor_collector` is `"COLLECTOR"`. This variable is used by DaemonCore for several purposes, such as constructing the name of this daemon's log file.

DaemonCore is an event-driven framework. All functions written by the daemon programmer must be *registered* as a callback to some event, or they will never be invoked. Every DaemonCore daemon *must* define the following handlers (callbacks), which do not need to be "registered" with DaemonCore:

**void main_pre_dc_init( int argc, char** argv)** Invoked by DaemonCore very early after the process is created. Rarely used – usually just an empty function.

**void main_pre_command_sock_init()** Invoked by DaemonCore early after the process is created, but after the configuration and logging subsystems have been initialized. Rarely used – usually just an empty function.

**int main_init(int argc, char **argv)** Invoked by DaemonCore after everything has been initialized. This function should be considered to be the `main()` function from the point of view of the daemon developer. DaemonCore will parse the command line and handle any parameters that are specific the DaemonCore (see section 3.8.2 of the Condor Manual), and then pass any remaining daemon-specific command line arguments to main_init(). In main_init(), typically command line arguments are parsed and, most importantly, callbacks for any interesting events that the daemon wishes to respond to are registered.

**int main_config( bool is_full )** When DaemonCore receives a `SIGHUP` signal (typically via the `condor_reconfig` command), it will reconfigure itself and then invoke this handler.

**int main_shutdown_graceful()** Invoked by DaemonCore when the daemon has been requested to shutdown gracefully, such as when the user issues a `condor_off` or a `SIGTERM` signal is delivered.

**int main_shutdown_fast()** Invoked by DaemonCore when the daemon has been requested to shutdown as quickly as possible, such as when the user issues a `condor_off` with the `-fast` argument, or a `SIGQUIT` signal is delivered.

So a typical DaemonCore-based daemon will register a bunch of event handlers in `main_init()`, and then the bulk of the rest of the code will be functions that respond to the registered events. Every registered event handler is expected to *return quickly to DaemonCore*. Because DaemonCore is (currently) single-threaded, any event handler that takes a long time to perform its duties will in block the entire daemon, possibly causing clients to timeout, etc. That means an event handler ideally should not make any blocking system calls, such as waiting for a network reponse – instead, the event handler could turn around and register another event before returning.

For every type of event handler, the programmer must provide a pointer to the either the C++ callback method or the C callback function (i.e. a function that is in the global scope, and not a member of an object) to be invoked when the event is triggered. The C++ callback type defintions will have a ”cpp” appended to their typenames. For instance, use the type definition of `TimerHandercpp` to register a timer handler that is a C++ method, and use `TimerHandler` to register a timer handler that is a C function or a static C++ method. When registering C++ methods with DaemonCore, it is required that the class containing the callback methods be derived from the empty class `Service`.

In addition to a function pointer, several DaemonCore event registration methods allow the programmer to provide an *authorization level*, such as READ, WRITE, ADMINISTRATOR, etc. Clients can be granted access to certain authorization levels via the settings in the configuration file as described in the Condor Manual, e.g. ALLOW_READ, ALLOW_WRITE, DENY_READ, DENY_WRITE, etc.

The following sections will give a brief overview of the types of events handled by DaemonCore.

## 3.1 Command Handlers

DaemonCore `command handlers` enable a simple model for remote method invocation.

By default, a DaemonCore daemon will be born with two listen sockets created at startup. One socket will be a ReliSock (TCP), and the other will be a SafeSock (UDP). Both sockets will share the same port number, which will be chosen dynamically unless otherwise specificed via command-line option or via a parameter passed into `Create_Process()` by the parent process.

When a connection is made by a client to this listen socket, DaemonCore will `accept` the connection if it is TCP, and then it will read one one integer off of the socket via CEDAR. This integer is refered to as a ”command” integer, and DaemonCore then looks up in its data structures to see if any command handler function has been registered to respond to this command int. If so, DaemonCore then checks the authorization level of the handler and compares it to the authorization level allowed by this client by calling the `Verify` method. This method return whether or not the client is authorized to perform a specified command given the client’s ip address and/or authentication information. If the authorization check fails, the connection is closed. If it passes, then the registered command handler is invoked.

The command handler is given access to the network socket, and is responsible for reading any subsequent parameters off of the wire and also for writing any results. Once the command handler returns, DaemonCore will close the socket (if it is TCP) unless the constant `KEEP_STREAM` is returned from the command handler. The idea here is all DaemonCore handler should attempt to return quickly. Command handlers could perform a non-blocking read by invoking the method `Register_Socket` (see section 3.3) on the socket passed into the handler in order to register a callback function to be invoked when there is more data to read on the socket, and then returning `KEEP_STREAM`.

See `Register_Command()` and related calls in the header file.

## 3.2 Signal Handlers

Signal handlers in DaemonCore can be thought of as commands that do not have any input or output parametes. DaemonCore methods exist to register a signal handler, as well as block and unblock signals. The method for sending signals in DaemonCore is `Send_Signal()`; signals may only be sent to a process' parent or children.

Signals are implemented in DaemonCore as follows: if a process is sending a signal to itself, just some in-memory data structures are manipulated. If a signal is being sent to a parent or child process and that process is linked with the DaemonCore library, then DaemonCore sends a command encoding the signal number to the command socket of the destination.

The UNIX operating systems have signal support in the operating system. Because DaemonCore delivers signals via its own mechanism, DaemonCore applications are not limited by the set of 20 or so signals provided by UNIX. Any number of signals can be defined; see file `src/condor_includes/condor_commands.h`. Furthermore, on UNIX, DaemonCore catches common POSIX operating system signals, such as SIGTERM, and turns around and raises that DaemonCore signal via `Send_Signal()`; then the UNIX signal handler returns. The actual DaemonCore signal handing function is then invoked later from the main driver loop. This means that DaemonCore signal handlers are not called from inside a UNIX signal handler, and thus are free from the hassles normally associated with UNIX signal handlers (such as being limited to only POSIX signal re-entrant function, volatile data, inability to call malloc, etc). For this reason and several others such as the lack of POSIX signals in Win32, programmers should not use UNIX/POSIX signal handling calls (like sigaction, sigprocmask, kill) in their DaemonCore programs.

The following signals are handled automatically by DaemonCore and therefore the programmer should not associate a handler with these signals:

**SIGTERM** Request a graceful shutdown; same as calling `Shutdown_Graceful()`.

**SIGQUIT** Request an immediate shutdown; same as calling `Shutdown_Fast()`.

**SIGSTOP** Request the operating system to suspend the process; same as calling `Suspend_Process()`.

**SIGCONT** Request the operating system to continue a previously suspended process; same as calling `Continue_Process()`.

**SIGHUP** Request a process to reconfigure itself (i.e. re-read its configuration).

**SIGKILL** Request the operating system to immediately destroy a process. The programmer should never send this signal to another process.

**SIGCHLD** A child process has exited; DaemonCore will determine which one and invoke the proper reaper handler.

See `Register_Signal()`, KeywordSend_Signal(), and related calls in the header file.

## 3.3 Socket Handlers

The programmer can register a handler that will be invoked whenever there is data ready to be read on a given CEDAR socket via the method `Register_Socket()` and related calls.

Similarly, if the non-blocking flag is passed to the ReliSock `connect()` method, then `Register_Socket` can also be used to invoke a handler once a TCP connection has been established.

## 3.4 Pipe Handlers

DaemonCore contains methods to create and destroy unamed pipes as a way to facilitate interprocess communication. DaemonCore pipes can communicate with other DaemonCore processes or with non-DaemonCore linked processes. A pipe created by the DaemonCore `Create_Pipe()` method can be passed to `Register_Pipe` in order to receive a callback whenever there is data ready to read on the pipe.

One reason DaemonCore needs its own methods to handle pipes is to deal with Win32. On UNIX, pipe descriptors and socket descriptors are fairly intechangable; both can be passed into `select()` in order to determine when they have data available. But on Win32, pipe descriptors cannot be passed into `select()`. Thus, in the internals of the DaemonCore implementation, when a pipe handler is registered, DaemonCore will pass the descriptor into the Win32 `WaitOnMultipleObjects()` system call which it invokes in another thread seperate from the thread that blocks on `select()`.. The implementation is further complicated by the fact that this Win32 system call can only watch a maximum of 64 objects. To get around this limitation, DaemonCore will start additional threads as needed so this limitation in Win32 is hidden from the DaemonCore developer.

## 3.5 Timer Handlers

Any number of timers may be registered with DaemonCore. The `Register_Timer()` method takes a pointer to a timer handler function and an integer that specifies how many seconds to wait until invoking the handler.

If a *periodic* value is specified, then the timer is automatically reset for the specified number of seconds when the timer handler returns. For example, the below will cause MyClass-¿foo() to be invoked in 5 seconds, and then automatically reset for every 30 seconds once foo() returns::

```
daemonCore->Register_Timer(5,30,(TimerHandlercpp)foo,MyClass);
```

Timers can be cancelled or reset (to a different time) at any point, including from within the timer handler function itself.

Realize that timers, like everything else in DaemonCore, are not preemptive. Thus, a periodic timer set to go off every 30 seconds may fire late if some other handler takes more than 30 seconds to return back to the DaemonCore driver. Furthermore, DaemonCore timer implementation is via a timeout value passed into `select()` in the main driver loop. Thus timers do not rely upon UNIX's SIGALRM facility, and timer handlers are not called from within a UNIX signal handler.

See `Register_Timer()`, `Reset_Timer()`, and related calls in the header file.

## 3.6 Reaper Handlers

A *reaper* is a handler that will be invoked when a process or thread terminates. The reaper handler will be passed the process or thread id of the process or thread that exited, as well as the exit status.

A reaper handler is registered via the `Register_Reaper()` method, which returns a reaper id ( a positive integer starting at 1). This reaper id is then passed as a parameter to `Create_Process()` or `Create_Thread()`.

DaemonCore will guarantee that the process id (pid) of a process that terminated will *not* get reused by the operating system until the reaper handler for that pid has returned back to the DaemonCore driver. This is not such a big deal on UNIX, since UNIX goes out of its way to procrastinate the reuse of process and thread ids as long as possible. However, this semantic guarantee is very important on Win32, since Win32 tends to reuse pids almost immediately after the handle to the process is closed.

The implementation of the reaper callback mechanism inside of DaemonCore is very tricky and was difficult to get correct, especially on Win32. On Win32, process handles are watched via calls to `WaitForMultipleObjects()` in another thread. DaemonCore can watch over and reap more than 64 child processes by spawning additional threads as needed - see section 3.4 for more discussion of this mechanism implemented inside of DaemonCore for monitoring non-socket Win32 handles.

See `Register_Reaper()` and related calls in the header file.

# 4 Process Management

Because process creation and management is so different in POSIX versus Win32, DaemonCore abstracts away these differences by providing its own methods for the creation, destruction, suspension, and testing for the existence of processes, as well as its own cross-platform implementation of POSIX-like functions such as `getpid()`, `getppid()`, `WIFSIGNALED()`, and many others.

The implementation details of a few of these facilities deserves special mention, and is discussed in the sections below.

## 4.1 Process Creation

The `Create_Process()` method spawns a child process (i.e. both a fork and exec on POSIX systems). Method parameters can control many aspects of process creation, such as the specification of the environment, redirection of stdio, operating system priority (nice value), which CEDAR sockets should be inherited by a child process, and many others. A few tricky implementation details of `Create_Process()` deserve special mention.

First of all, internal to the implementation on POSIX, a temporary pipe is created between the parent and child process. This pipe is used to pass back any errors that may occur *after* the call to `fork()` and *before* the completion of the call to `exec()`. This is a nasty problem with POSIX that DaemonCore hides from the developer.

Another implementation detail: if the process to be created is a DaemonCore process (i.e. linked with the DaemonCore library), then it is the *parent* process that creates the command sockets. The child simply inherits these sockets from the parent. The reason for this is it allows the parent to intiate the sending of signals and commands to the child before the child spawn process has completed, because the parent knows the IP port of its child process a priori. This eliminates the nasty situation where the parent wants to send a signal to its child before the child has told the parent the address of its command port.

Another facility provided ”behind the scenes” in DaemonCore is the automatic killing of a child process that appears to be hung. This fits the model of Condor, where every daemon is responsible for the management and cleanup of any child processes it may spawn. How it works is when a DaemonCore process starts, it looks to see if its parent process is a DaemonCore process. If so, then it sends its parent a heartbeat command at a specified interval (can be specified per subsystem in the configuration file). Once the parent process receives a hearbeat, it will hard kill the child process if subsequent heartbeats do not arrive before a timeout. The DaemonCore user can call method `Was_Not_Responding()` in the reaoper callback to determine if a child exited of its own accord or was forcibly killed by DaemonCore because it appeared to be locked up. Because the parent will *do nothing until the first heartbeat has been received*, non-DaemonCore processes may be safely spawned without fear that DaemonCore may kill them because no heartbeat is sent.

## 4.2 Process Suspension

The `Suspend_Process()` method on UNIX is trivial, but on Win32 it is tricky because Win32 only provides a primitive to suspend a specific thread within a process, not the entire process itself. Therefore, the implementation becomes complicated because it must handle problems that arise from a lack of an atomic process suspend on Win32 – it is not sufficient to simply iterate through all the threads in the process and suspend them.. For example, consider the following: we want to suspend a process that has two threads, A and B; we suspend A; then, before we can suspend B, thread B sends a continue to thread A; we then suspend thread B. Now we are left in a state where thread A is still active. The DaemonCore implementation handles these situations.

### 4.3 Thread Creation

DaemonCore includes a method named `Create_Thread()`. This method is best ignored, and should ultimately be depricated, because in its current incarnation it does vastly different things on UNIX -vs- Win32. On UNIX, this method starts a new process via `fork()` but does not call `exec()`. Because there is no equivelent to a fork call on Win32, this method when invoked on Win32 actually starts a new kernel thread. Because the vast majority of the code in Condor is not thread safe, this is a dangerous thing to do. Thus most of the currently existing calls to `Create_Thread` are located in places where functionality is restricted to UNIX.

## 5 Logging

The API to a simple debug console logging interface is primarily via the `dprintf()` function. `dprintf()` is just like good 'ol `printf()`, except that it also takes a logging *subsystem* parameter. The available logging subsystems are defined in `condor_debug.h` and examples include `D_SECURITY`, `D_DAEMONCORE`, and `D_FULLDEBUG`.

The logging of different subsystems can be enabled or disabled via the configuration file. All messages are written into a specified file that can be automatically (and safely) rotated once it reaches a specificed size.

Different subsystems can have their messages duplicated into different files, each with their own file rotation size – for example, all messages for the `condor_startd` can be logged into the file `StartLog`, and in addition all `D_SECURITY` messages can be logged into a second file `StartLogSecurityEntries` that never rotates.

The configuration file can specify that the logging system in DaemonCore obtain an exclusive write lock on the file before writing to it. This is handy if multiple daemon processes want to write into the same log file, as is typical with the `condor_shadow`. The actual lock file can be specified to exist in a different location than the log file, enabling the log files to reside on a shared filesystem like NFS and still have proper locking semantics.

## 6 Configuration

## 7 Client Library

## 8 Future Directions

This section briefly presents a few of the major changes in store for DaemonCore, currently in either the planning stages or early implementation phases at UW-Madison.

### 8.1 Threads

A significant gain in scalability of `condor_c`ould be realized if DaemonCore daemons could easily overlap computation and I/O. While that can be done with the current implementation via the generous use of `Register_Socket()`, this callback model is hard for programmers to work with. Therefore, in the later half of 2004, the Condor Team and the University plans to enhance DaemonCore to support non-preemptive cooperative threads. Such a system would only preempt a thread when the thread explicitly relinquishes control via a yield() call. The plan calls for threads to yield solely when they would otherwise block on I/O, or when explicitly coded to do so at "safe" moments.

This approach is likely the easiest and safest way to introduce threading to a large body of existing code that was not originally designed to be thread-safe. Furthermore, this approach should be easier to maintain into the future considering that (i) the Condor Team has a large amount of developer turnover of various skill levels due to the continuous flow of graduate students that join and leave the team; and (ii) a small mistake in a preemptive threading model can easily result in a non-deterministic bug that may be extremely difficult to reproduce.

### 8.2 SOAP

# 9 Conclusion