

*Net Integration Technologies, Inc.*



# **WvStreams**

**An Easier Way to World Domination**

Dave Coombs  
v1.0 - Release: 2003 09 26

**FOR PUBLIC RELEASE**

# TABLE OF CONTENTS

<b>1</b>	<b>Informative Introduction.....</b>	<b>3</b>
<b>2</b>	<b>Why We Wanted WvStreams.....</b>	<b>3</b>
2.1	Prevent Programming Problems.....	3
2.2	Have Hope, Hiding Hideous Hacks Helps!.....	4
2.3	Functional Feats, Fantastically Fast.....	4
<b>3</b>	<b>Startling Stunts.....</b>	<b>4</b>
3.1	WvStreams Works Wonders.....	4
3.2	Trivial TCP/IP.....	5
3.3	Crazy Cryptography.....	5
3.4	Compelling Compression.....	5
3.5	Driving Down Directories.....	5
3.6	Cunning Configuration.....	6
3.7	Insanely Ingenious IPC.....	6
3.8	Painless Portability.....	6
3.9	Unbelievably Useful UrlPool.....	7
3.10	Fabulously Fancy FAM.....	7
3.11	Oggs Over-easy.....	7
3.12	Multitasking Mayhem.....	8
<b>4</b>	<b>Technical Tips &amp; Tutorial.....</b>	<b>9</b>
4.1	Start Studying Streams.....	9
4.2	Base-class Behaviour.....	9
4.3	Some Simple Streams.....	10
4.3.1	WvFile.....	10
4.3.2	WvPipe.....	11
4.3.3	WvTimeStream.....	11
4.4	Nifty Networking.....	12
4.4.1	WvTCPListener.....	12
4.4.2	WvTCPConn.....	12
4.4.3	WvUDPStream.....	12
4.5	WvStreamList.....	13
4.6	Easy Elegant Encoders.....	13
4.6.1	Consecutive Coder Combinations .....	14
<b>5</b>	<b>Crazy Creations.....</b>	<b>14</b>
5.1	Cryptographic Conversation Connector.....	15
5.2	Streaming Sound Server.....	16
<b>6</b>	<b>Future Fearless Frontiers.....</b>	<b>17</b>
<b>7</b>	<b>Confusing Conclusion.....</b>	<b>17</b>

# 1 INFORMATIVE INTRODUCTION

WvStreams is an open-source C++ networking library developed over the last few years with the following major goal: make coding easy, without sacrificing performance. Really.

In this paper, we'll discuss:

- The various things that WvStreams is really good at, and exciting ways that you can benefit from them.
- Things we've done with WvStreams, to prove that we're not crazy. (At least, not the bad kind of crazy.)
- Some of the more useful areas of WvStreams, and broad generalizations about how to use them.
- Some wacky examples of things that would ordinarily be really complicated. With WvStreams, they're so easy I'm including the code.
- Exactly how happy we'd be to hear from you about WvStreams, if you're working *with* it or *on* it.

## 2 WHY WE WANTED WvSTREAMS

... and other alliterations

### 2.1 Prevent Programming Problems

WvStreams came about because we wanted to write a really big C++ application to control an entire Linux system acting as an Internet gateway. We decided to abstract away the most common parts of programming by making a library that would act as the framework for our main project, but which we could use in future projects too. Do the hard stuff right once, and then never worry about it again.

So the question is: what *do* programmers really need? What sort of stuff do we do all the time, that this library should encompass?

- File I/O. We all read and write files all the time.
- String manipulation and buffers. Admit it, you *still* do this wrong sometimes.
- Maintain sets of things. Have you written a linked list or a hash table? Have you done this more than once? More than twice? I thought so.
- Networking. We need to make and accept TCP connections, send and receive UDP packets, and keep track of multiple instances of these.
- Configuration. Most programs need to store their config options somewhere, and most programmers are too lazy to care if their config system is ugly or inflexible.
- Multitask. Threads are dangerously evil. What to do?

## 2.2 Have Hope, Hiding Hideous Hacks Helps!

Some of the above things are really hard to do properly. That's why programmers so often cheap out on their config systems, string manipulation, and buffer management, even though those things are important to get right.

That's what WvStreams is really good at. It does most of the hard stuff for you, and presents a clean, friendly API. It takes a bit of getting used to, but we've put lots of work into making the underlying code solid, and we think it was worth the effort. Most people who have seen WvStreams-based code comment to us on how readable it is, and are surprised to find how easy it is to write working programs with it.

We have a philosophy about Justifiable Ugliness: "Any amount of code ugliness is okay if it removes more ugliness than it adds." The insides of WvStreams are quite complex, but using it to write clean, readable, fast code is easy.

## 2.3 Functional Feats, Fantastically Fast

To illustrate that we are not lying, the following are open-source projects we've developed using WvStreams:

- Tunnel Vision, a simple, secure, multi-node VPN system, which shares routing and DNS info, was written in one weekend.
- Retchmail, "the world's most stupidly fast POP3 retriever", was written in *one evening*. It retrieves multiple messages at the same time, in increasing order of size. (The FAQ on our website explains why this makes it stupidly fast.)
- WvTftp is the world's fastest TFTP server. Our tests showed it to be 3.5 times faster than other TFTP servers. That may not sound exciting, but maybe you don't netboot very much.
- I am compelled to mention WvDial, our most popular application, even though it was written before WvStreams grew most of the useful bits it has today. The WvDial code isn't great, and it could really benefit from WvBuf and WvCont. See? I *wish* I had WvStreams back then!

WvStreams is *pragmatic*. It's carefully optimized to make you write the fewest possible lines of code to do common tasks. `wvStream::getline()` works just like you want it to work. The standard C++ `iostream::getline()` doesn't.

# 3 STARTLING STUNTS

## 3.1 WvStreams Works Wonders

This section is specifically about what WvStreams can do to make your life and/or projects easier. Since WvStreams is designed to let you do common, complicated tasks using almost no code, this should be of interest.

You don't need to understand *everything* in the code snippets that follow. The point is that they're small, easy to read, and they actually work. If you do want to figure out how they work, you can read section 4, or even the source code to WvStreams itself.

So: What *can* we do for you, anyway?

## 3.2 Trivial TCP/IP

The main thing we wanted from WvStreams was an easy way to write network code. We had done plenty of sockets programming, got annoyed, and wanted to abstract it into something trivial. The following code, we think, is trivial.

```
WvTCPConn *conn = new WvTCPConn("www.yahoo.com", 80);
conn->noread();           // not gonna read
conn->print("Chickens have feathers!\n");
conn->nowrite();         // done writing now
WvIStreamList::globallist.append(conn, true);
```

Here we connect to TCP port 80 of www.yahoo.com, and talk about poultry plumage. Yahoo sysadmins love this. When the stream is done sending data, it is automatically deleted. If you write network programs any other way (especially with sockets), you're working too hard.

## 3.3 Crazy Cryptography

Easy. Want to encrypt a message with Blowfish, and *then* send it to Yahoo?

```
WvTCPConn *conn = new WvTCPConn("www.yahoo.com", 80);
WvBlowfishStream *fish = new WvBlowfishStream(conn,
                                                (const void *) "secretkey", 9);

fish->noread();
fish->print("Chickens have feathers!\n");
fish->nowrite();
WvIStreamList::globallist.append(fish, true);
```

All we're doing is wrapping the TCP connection inside a Blowfish "clone" stream, and then writing to that. If Yahoo was smart enough to send back an encrypted reply, `fish->getline()` would decrypt it for us.

## 3.4 Compelling Compression

This works the same way as encryption, really, which is to say that it's really easy. I hereby present to you the world's most hilariously short implementation of `zcat`:

```
WvFile *f = new WvFile("whatever.gz", O_RDONLY);
WvGzipStream *gz = new WvGzipStream(f);
gz->autoForward(*wvcon);
WvIStreamList::globallist.append(gz, true);
```

Here, we are telling the `WvGzipStream` wrapped around the `WvFile` to automatically forward everything it reads to `wvcon`, which is basically `stdout`.

## 3.5 Driving Down Directories

Traversing a directory tree is a pain in the butt. You have to use silly `opendir()`, `readdir()`, and `closedir()`, and if you happen to encounter a subdirectory, you have to `opendir()` that, and not lose track of what you were doing.

`WvDirIter` iterates through a directory tree, and takes care of the annoying recursion for you. Optionally, you can choose not to recurse, if you want.

Here's a ridiculously short implementation of "find /":

```
WvDirIter i("/", true);           // true means recurse
for (i.rewind(); i.next(); )
{
    printf("%s: size=%s\n",
           (const char *) i->fullname, i->st_size);
}
```

If there's a way to make that even easier, we couldn't think of it.

### 3.6 *Cunning Configuration*

UniConf is our universal configuration system. It's universal because it has pluggable back ends *and* front ends (hence you can configure KDE with gconf, or configure Gnome with KConfig), and we wrote it because every other config system sucks.

Reasons they all suck include poor API design, ugly config-file format, lack of notifications, lack of defaults or overrides, lack of proper caching, and more. UniConf is designed to get all of these right.

For much more information, please read Avery Pennarun's UniConf paper from Gu4dec, available at <http://open.nit.ca/unicconf.pdf>

### 3.7 *Insanely Ingenious IPC*

Inter-Process Communication is one of those things that everybody loves to hate. Of course you want it to be fast, safe, and easy to use. Pick any two, as they say.

With WvMagicCircle, you can have all three. It's a simple circular buffer that resides in shared memory. You can create a WvMagicCircle, and then `fork()`, and then send messages between your two processes `get()` and `put()`. Watch:

```
WvMagicCircle q(1024);
if (fork() > 0) {
    // parent
    q.put("some stuff", 10);
    ...
} else {
    // child
    while(1) {
        char buf[1024];
        q.get(buf, 1024);
        printf("%s\n", buf);
    }
}
```

It's fast because reading and writing shared memory doesn't involve any slow system calls. It's safe because you always write to the end and read from the beginning. And it's easy to use, because we couldn't find a way to make it hard. Sorry.

### 3.8 *Painless Portability*

The `main()` part of any WvStreams program is almost always an event loop. You make a bunch of streams, add them to a WvStreamList, and loop forever on your list, running whichever streams are ready at any given time. Somewhat conveniently, other systems, particularly GUIs, follow a similar design.

We have created WvStream wrapper libraries for Qt, Gtk, and even Win32. (Guess which one was the most work?) This means you can take ordinary WvStream objects, and stick them, via these wrappers, inside a Qt event loop, or a Gtk event loop, or a Win32 event loop, and your streams will magically work. We have written KDE programs, Gnome programs, and a plugin for Microsoft Outlook this way.

### 3.9 *Unbelievably Useful UriPool*

Need to download a file in your program? Meet WvUriPool, a fast, easy-to-use, parallelizing, pipelining HTTP/1.1 and FTP file retriever. How does it work? It works like this:

```
WvUriPool *pool = new WvUriPool();
WvStream *s1, *s2, *s3;
s1 = pool->addurl("http://somewhere.com/somefile");
s2 = pool->addurl("http://elsewhere.org/picture.jpg");
s3 = pool->addurl("ftp://ftp.moo.com/cowsong.mp3");
mystreamlist.append(pool, true);
```

You make a pool, add some URLs to it, and it gives you a bunch of streams in return. It makes as many connections as it needs to, uses HTTP 1.1 keepalive, and gives you a stream for each file. You can `autoforward()` each stream to a WvFile to save it if you feel so inclined.

As an exercise, I urge you to write a web browser using WvUriPool and our Gtk event-loop integration.

### 3.10 *Fabulously Fancy FAM*

FAM is the File Alteration Monitor, which is, trust me, something you've always wanted. We certainly always wanted it. But it's not as easy to use as it should be. Our WvFAM wrapper definitely is.

```
void fam_cb(WvStringParm name, WvFamEvent e, bool dir)
{
    wvcon->print("%s changed\n", name);
}

WvFam f(fam_cb);
f.monitor("/etc/passwd");
f.monitor("/etc/group");
```

Your program will print a message whenever either `/etc/passwd` or `/etc/group` is written to.

One of our projects is a distributed filesystem based on this.

### 3.11 *Oggs Over-easy*

Ogg Speex and Ogg Vorbis are great open-source libraries for compressing audio streams. We have wrapped these libraries into our super-convenient WvEncoder design.

For our office, we designed and wrote our own phone system. When someone calls in, hits my extension, and leaves me a voice message, the message is Vorbis-encoded, base64-encoded, word-wrapped, and *emailed* to me.

Here's the code that does the encoding. Note how long it isn't.

```
WvFile *msgfile = new WvFile(filename, O_WRONLY|O_TRUNC|O_CREAT);

WvEncoderStream *encstream = new WvEncoderStream(msgfile);
encstream->auto_flush(false);
WvOggVorbisEncoder *oggenc = new WvOggVorbisEncoder(
    WvOggVorbisEncoder::VBRQuality(1.0), SND_RATE, 1);
encstream->writechain.append(
    new WvPCMSigned16ToNormFloatEncoder(), true);
encstream->writechain.append(oggenc, true);
oggenc->add_comment(WvString("VoiceMail from: %s", "someone"));
encstream->writechain.append(new WvBase64Encoder(), true);
encstream->writechain.append(new WvWordWrapEncoder(72), true);
```

Take the 16-bit PCM voice message you recorded, write it to `encstream`, stick some MIME headers around that, and off it goes.

## 3.12 Multitasking Mayhem

Making sure code is thread-safe is kind of like juggling piranhas -- it's difficult, dangerous, and it'll probably bite you. However, it's both possible and incredibly fun to write a co-operative task-switcher using the `setjmp()` and `longjmp()` system calls, in POSIX-compliant, portable C. So we did. You don't have to worry about any nasty preemption, mutexes, races, etc.

`WvCont` is a wrapper class for an arbitrary callback function. Inside the callback, you can voluntarily `yield()`, which makes your function return. Next time the callback is called, it is as if `yield()` has returned, and you keep running. Here's a sample program:

```
static void *cb(int i, void *)
{
    printf("Entering %d.\n", i);
    WvCont::yield(NULL);
    printf("I'm back! Leaving %d.\n", i);
    return NULL;
}

int main()
{
    WvCont c1(WvBoundCallback<WvCont::Callback, int>(cb, 42));
    WvCont c2(WvBoundCallback<WvCont::Callback, int>(cb, 4242));

    c1(NULL); c2(NULL); sleep(1);
    c1(NULL); c2(NULL);
}
```

And here's its output:

```
Entering 42.
Entering 4242.
I'm back! Leaving 42.
I'm back! Leaving 4242.
```

Note how both tasks print the entering message, and then both tasks print the leaving message. The only thing between those two print statements in the callback is the call to `yield()`.

# 4 TECHNICAL TIPS & TUTORIAL

This section is a quick-and-dirty summary of the important points you should know in order to write your own WvStreams code. It is by no means complete, but it should be enough to get you going. If you get bored, stop reading and write something.

## 4.1 Start Studying Streams...

In WvStreams, nearly everything is a stream, and these are combined into lists of streams. Program flow is based on which streams in the lists are ready to do things at different times. They all inherit the same basic API, but there are broadly two categories of streams: those that are read and written when possible, and those that are called to do something when they indicate they're ready.

Generally, the main event loop in a WvStreams program looks like this:

```
...initialize stuff
WvIStreamList::globallist.append(some_stream);
WvIStreamList::globallist.append(some_other_stream);
...
while (WvIStreamList::globallist.count())
{
    l.select(-1);
}
```

`select()` tells you if a stream is ready to be operated on, or has data available to be read. On a list, the `select()` and `callback()` calls are distributed to all streams in the list.

## 4.2 Base-class Behaviour

The WvStream base class doesn't do much on its own (it doesn't even necessarily have a file descriptor associated with it) but it does provide the API that all other stream classes conform to.

Here are the fundamental parts of the API. There are other functions that aren't listed, but these are the truly important ones:

```
bool isok() const;
void close();
size_t read(void *buf, size_t count);
size_t uread(void *buf, size_t count);
size_t write(const void *buf, size_t count);
size_t uwrite(const void *buf, size_t count);
bool isreadable();
bool iswritable();
char *getline();
bool select(time_t msec_timeout, bool r, bool w);
bool continue_select(time_t msec_timeout);
void alarm(time_t msec_timeout);
void setcallback(WvStreamCallback func, void *data);
void execute();
```

The `isok()` and `close()` functions don't need describing. Reads and writes are buffered automatically with `WvBuf` for efficiency, which is the difference between `read/write` and `uread/uwrite`. The 'u' means unbuffered. You would call the buffered functions to read and write from a stream, but you would *override* the *unbuffered* functions to change what a derived stream class does. Note that `read` and `write` also have variants that take a `WvBuf` rather than a void pointer.

The `isreadable()` function is true either if there is data in the input buffer available for reading, or if `select()` for readability returns true. Similarly, `iswritable()` is true if `select()` for writability returns true.

Now, `getline()` is cool precisely because we use a `WvBuf` to buffer the data that's read from the stream. It'll pull data out of the buffer until the first newline and return a pointer to the line of text. If there's no newline in the buffer, it returns `NULL`. This is very handy, and much easier than doing it yourself.

The `select()` function defaults to selecting for read and not for write, but you can override that with the `bool` arguments. It will return true if your stream is ready for read and/or write, within the given timeout. If the timeout is -1, it waits forever. If 0, it returns immediately (`isreadable` and `iswritable` call this). If greater than 0, it waits that long for the stream to become ready, before giving up.

`continue_select()` works similarly to `select()`, but is crazy. It uses `WvCont`, as described in section 3.12. Basically, calling `continue_select()` from your callback or `execute()` function is like returning from it, but next time your function is called, you come back to where you returned from. And it has a timeout, so even if nobody calls you, you'll get called.

`alarm()` forces your stream to be readable, for exactly one `select()` call, after the given time delay. Use this to make your callback get called after a certain delay.

`setcallback()` is used to define what your stream will do when it becomes ready. Pass it a function pointer and another optional piece of data, and your function will be called with that data as an argument. If you don't set a callback, the overridable `execute()` function is called instead when your stream becomes ready.

Now that you know generally how a `WvStream` is used, let's look at a few particular implementations.

## 4.3 Some Simple Streams

### 4.3.1 WvFile

`WvFile` is probably the simplest stream that's actually useful. It takes the `WvStream` interface, and binds it to a file descriptor returned from `open()`. You can `read()` from a `WvFile`, `write()` to a `WvFile`, and most importantly, `getline()` from a `WvFile`. Here's a simple example:

```
WvFile f("/etc/profile", O_RDONLY);
if (f.isok()) {
    char *line;
    while ((line = f.getline(-1)) != NULL)
        printf("profile says: %s\n", line);
    f.close();
}
```

### 4.3.2 WvPipe

WvPipe forks and execs an arbitrary program, directing the stream's output to the program's stdin, and getting the stream's input from the program's stdout.

WvPipe has additional functions to determine the program's exit code (once it finishes and goes `!isok()`), and to determine the pid of the child process.

Here's a cheesy example that shows reading, at least:

```
const char *argv[] = { "ls", NULL };
WvPipe p(argv[0], argv, false, true, true);
if (p.isok())
{
    printf("pid is %d\n", p.getpid());
    char *line;
    while ((line = p.getline(-1)) != NULL)
        printf("ls says: %s\n", line);
    p.close();
    printf("exit code was %d\n", p.exit_status());
}
```

Here, we passed three extra flags to WvPipe's constructor. They are writable, readable, and catch\_stderr, respectively. In this case, we weren't going to write to `ls`'s input, but we were going to read its output, and we might as well also read any errors.

Try changing `argv` to `{ "ls", "/foo", NULL }`, or something else that doesn't exist, and watch the sparks fly.

### 4.3.3 WvTimeStream

Unlike the previous two, WvTimeStream is a callback-when-ready stream, rather than a read/write stream. The general idea is as follows:

```
WvTimeStream ts;
ts.set_timer(100);
ts.setcallback(do_something, NULL);
```

This basically ensures that the `do_something()` callback will be called, on average, 10 times per second. (`set_timer()` takes a delay in milliseconds.)

Of course, this won't work unless something is actually running `select()` on the time stream. Normally this would be a WvStreamList, as described below in section 4.5.

Since we can't guarantee that the time stream will be selected and have its callback run exactly on time, WvTimeStream tries to compensate for this. For a simplified example, if we are selected and get our callback 250 milliseconds after the previous event, because the program was busy doing something else, we would actually get two callbacks in a row, and the third one would be scheduled for 50 milliseconds after that. It keeps an *average* of 10 callbacks per second in this example.

## 4.4 Nifty Networking

The most important kinds of networking streams are `WvTCPListener`, `WvTCPConn`, and `WvUDPStream`. For the most part, they work just like a regular `WvStream`, except for the TCP listener, which doesn't have to do as much.

### 4.4.1 WvTCPListener

The `WvTCPListener` is actually a callback-when-ready stream. When a connection request comes in, you get a callback, and your callback should run the `accept()` method, which returns a new `WvTCPConn`.

It is difficult to give a trivial example of this, but it is well illustrated in the *ircctest* sample program given in section 5.1.

### 4.4.2 WvTCPConn

There are two ways to make a `WvTCPConn`. As a server, `wvTCPListener::accept()` returns one. To initiate a connection, on the other hand, pass the IP address (or hostname) and port number of the destination, as was shown in section 3.2.

We used the `globallist` there because we don't want to wait for the DNS resolution and the actual TCP connection to take place. It'll connect eventually (the string we printed is buffered), and since we shut down the read and write file descriptors, the stream will automatically be removed from the `globallist` and deleted when it is finished.

### 4.4.3 WvUDPStream

Using `WvUDPStream` is about the easiest way to write UDP code. It takes a local `WvIPPortAddr` and a remote `WvIPPortAddr`, and lets you read and write UDP packets to and from the remote address.

If the remote address is `0.0.0.0`, the UDP socket is not connected, so packets can be received from anywhere, and sent to anywhere. As with the rest of `WvStreams`, this makes it easy to just do what you want, without worrying about the details. Note that you should be careful with `WvStreams` buffering, especially in non-connected mode, since UDP doesn't guarantee delivery.

```
WvUDPStream u("0.0.0.0:49152", "0.0.0.0:0");
u.setcallback(udp_echo, NULL);

static void udp_echo(WvStream &s, void *)
{
    char *line = s.getline(0);
    if (line)
        s.print("You said %s\n", line);
}
```

This works because writing to a `WvUDPStream` always sends to the address it most recently received from.

## 4.5 *WvStreamList*

`WvStreamList` is magic. It's both a `WvStream` as well as a linked list of other `WvStreams`, and is generally used to control the flow of a `WvStreams`-based program. It has the following properties:

- It can contain any type of `WvStream`, including other `WvStreamLists`.
- When `select()` is run on a `WvStreamList`, it recursively runs `select()` on all members of the list, returning true if any members are ready for a callback.
- When `callback()` is run on a `WvStreamList`, it runs `callback()` on any members of the list that were ready in the last `select()`. Think about that.
- When a member of a `WvStreamList` becomes `!isok()`, it is forcibly removed from the list, and optionally auto-deleted.

Usually the main body of a `WvStreams` program is just a `select` loop on a list of streams. You can put anything in the list, and get as complicated as you like. It often looks something like this, only less made-up:

```
WvTimeStream ts();
ts.set_timer(5000);           // every 5 seconds
ts.setcallback(do_something, NULL);
WvFancyStream f;             // its execute() does something
WvDancyStream *d = some_pointer;

WvStreamList list;
list.append(&ts, false);
list.append(&f, false);
list.append(d, true);        // auto-free

while (f.isok() && !want_to_die) {
    if (list.select(-1))
        list.callback();
}
```

In this case, the program will terminate whenever the `WvFancyStream` closes, or whenever `want_to_die` becomes true, perhaps because of a signal. What makes the loop special, though, is that it's so easy, and that you don't have to block waiting for any particular stream to become ready. Whatever's ready gets called, each time through the loop.

For what it's worth, there's nothing stopping you from using `select(5000)` instead of `select(-1)`, and putting the contents of the timer's callback right in the main loop. I just felt bad for not including a `WvTimeStream` example two pages ago.

## 4.6 *Easy Elegant Encoders*

`WvEncoder` is a relatively new addition to `WvStreams`. A `WvEncoder`-derived class must define a function that takes an input `WvBuf`, and an output `WvBuf`. It reads from one, performs some transform on the data, and writes the results to the other `WvBuf`. (The `WvEncoder` base class provides numerous wrappers for encoding from a string, to a string, to an area of memory, etc.)

We currently have WvEncoders ranging from the mundane to the really, really cool. Here are some existing examples:

- WvBase64Encoder is useful for parsing MIME-encoded messages, or generating HTTP-authentication passwords.
- WvWordWrapEncoder inserts newlines between words in a text buffer so that no line is longer than a given number of characters, such as 80.
- WvGzipEncoder compresses or decompresses the input.
- WvBlowfishEncoder and WvRSAEncoder encrypt or decrypt the input buffer using a given key. (This uses OpenSSL, so you don't have to! Yay!)
- WvOggSpeexEncoder and WvOggVorbisEncoder use the standard Ogg libraries to encode or decode an audio stream.
- WvFFT consists of three encoders that do a forward Fast Fourier Transform on an input stream.

#### 4.6.1 Consecutive Coder Combinations

A WvEncoderChain is rather like a WvStreamList. It is itself a WvEncoder, and it contains a sequence of WvEncoders that it will apply, in order, to its input, to produce its output. You could create a WvEncoderChain that, given some text, automatically wordwraps it, then gzips it, then encrypts it with blowfish.

The great thing is that you could create this wordwrapping, gzipping, and blowfishing encoder in about 10 minutes, instead of having to twiddle with all the different libraries yourself.

Maybe your program already has a network protocol, but it's really slow. Why not wrap it in a WvGzipStream, or run a WvGzipEncoder on a buffer you already have?

Maybe you're concerned about security. Put a WvRSAStream on your connection, use it to negotiate a blowfish key, switch to a WvBlowfishStream, and run your protocol through that. That's how our Tunnel Vision VPN works. Or even just use a WvSSLStream.

These encoders are largely interchangeable, and very easy to use.

## 5 CRAZY CREATIONS

All of the sample code on the following pages is available here:

<http://people.nit.ca/~dcoombs/lk2003/>

## 5.1 Cryptographic Conversation Connector

This program accepts multiple SSL TCP connections, and then relays messages between them. It's like IRC, only secure, and without all the features. It's a good example of callbacks and WvTCPListener.

```
static WvX509Mgr * x509;

typedef WvBoundCallback<WvStreamCallback, WvStreamList &> MyCallback;

static void stream_bounce_to_list(WvStreamList &l, WvStream &s, void *)
{
    WvStreamList::Iter out(l);
    char *line;

    while ((line = s.getline(0)) != NULL)
        for (out.rewind(); out.next(); )
            if (&out() != &s && out().select(0, false, true))
                out().print("%s> %s\n",
                    s.src() ? (WvString)*s.src() : WvString("stdin"), line);
}

static void accept_connection(WvStreamList &l, WvStream &s, void *)
{
    WvTCPListener &listener = *(WvTCPListener *) &s;
    WvTCPConn *conn = listener.accept();

    if (conn) {
        WvSSLStream *ssl = new WvSSLStream(conn, x509, false, true);
        MyCallback bounce_cb(stream_bounce_to_list, l);
        ssl->setcallback(bounce_cb, NULL);
        l.append(ssl, true);
    }
}

int main(int argc, char **argv)
{
    WvString myname("peanut.internal.nit.ca");
    WvString dn("O=%s,OU=Weaver.%s",
        myname, encode_hostname_as_DN(myname));

    x509 = new WvX509Mgr(dn, 1024);

    WvLog log("ircctest"), err = log.split(WvLog::Error);
    WvStreamList l;
    WvTCPListener listener(WvIPPortAddr(argc==2 ? argv[1] : "0.0.0.0:0"));

    MyCallback bounce_cb(stream_bounce_to_list, l);
    MyCallback accept_cb(accept_connection, l);
    wvcon->setcallback(bounce_cb, NULL);
    listener.setcallback(accept_cb, NULL);
    log("Listening on port %s\n", *listener.src());

    l.append(&listener, false);
    l.append(wvcon, false);

    while (listener.isok() && wvcon->isok()) {
        if (l.select(-1))
            l.callback();
    }

    if (!listener.isok() && listener.geterr())
        err("%s\n", listener.errstr());
    if (!wvcon->isok() && wvcon->geterr())
        err("%s\n", wvcon->errstr());
    return 0;
}
```

## 5.2 Streaming Sound Server

Pretend for a moment that you have a microphone plugged into your computer, working ALSA drivers (ha!) and something useful to say. Suddenly, you have the world's cheesiest audio streaming system! Wow! Not bad for 30 minutes of coding.

```
static bool want_to_die = false;

static void handler(int sig)
{
    printf("Caught signal %d, going away...\n", sig);
    want_to_die = true;
}

static void accept_callback(WvStreamList &l, WvStream &s, void *)
{
    WvTCPListener &listener = *(WvTCPListener *) &s;
    WvTCPConn *conn = listener.accept();
    if (conn) {
        l.noread();
        l.force_select(false, true);
        l.undo_force_select(true, false);
        l.append(conn, true);
    }
}

typedef WvBoundCallback<WvStreamCallback, WvStreamList &> MyCallback;

int main(int argc, char **argv)
{
    WvLog log("enctest"), err = log.split(WvLog::Error);
    WvStreamList l;

    signal(SIGTERM, handler);
    signal(SIGINT, handler);
    signal(SIGHUP, handler);
    signal(SIGPIPE, SIG_IGN);

    WvTCPListener sock(WvIPPortAddr(argc==2 ? argv[1] : "0.0.0.0:0"));
    sock.setcallback(MyCallback(accept_callback, l), NULL);
    l.append(&sock, false);
    log("Listening on port %s\n", *sock.src());

    WvDsp dsp(500, 22000, 8, false, true, false, false, false);
    WvEncoderStream enc(&dsp);

    WvOggVorbisEncoder *oggenc = new WvOggVorbisEncoder(
        WvOggVorbisEncoder::VBRQuality(1.0), SND_RATE, 1);
    enc.writechain.append(new WvPCMSigned16ToNormFloatEncoder(), true);
    enc.writechain.append(oggenc, true);

    char buf[1024];
    while (sock.isok() && enc.isok() && !want_to_die) {
        if (l.select(-1))
            l.callback();
        size_t ret = enc.read(buf, 1024);
        WvStreamList::Iter i(l);
        for (i.rewind(); i.next(); ) {
            if (i->select(0, false, true))
                i->write(buf, ret);
        }
    }
    return 0;
}
```

## 6 FUTURE FEARLESS FRONTIERS

We are planning The Great WvStreams Inversion. A question we commonly get is "Why the heck are you polling everything?" (Running `select()`, even on a list of streams, constitutes polling, since we have to check to see if anything's ready.) We're going to very slowly and carefully turn everything inside out, and make WvStreams into an event-driven system, which we expect will improve efficiency.

Send us some email if you want to know more.

## 7 CONFUSING CONCLUSION

So the frog said to the squirrel, "Of course I like bananas, you idiot!"

WvStreams does a whole lot of hard stuff, so that you don't have to.

But if you *wanted* to..... you're welcome to help. You can get more information on WvStreams and access our anonymous CVS repository by visiting [open.nit.ca](http://open.nit.ca), or by emailing [dcoombs@nit.ca](mailto:dcoombs@nit.ca).

I can't tell you whether WvStreams is right for your project, since I have no idea what your project is. But you might find it helpful, so we'd be pleased if you'd consider it. There is a WvStreams mailing list, [wvstreams-dev@lists.nit.ca](mailto:wvstreams-dev@lists.nit.ca), and we'd be very happy to hear about your experiences and questions working with WvStreams.

Very, very happy. Really.

Thanks!