Crosstalk

A system for plain-text request/response processing

Andrew Weinrich

March, 2003

Introduction

Overview

This paper describes Crosstalk, a Perl system for providing services based on plain-text requests and responses. Unlike other Instant Message- or Web-only systems, which are limited both in their external interface and the services they provide, Crosstalk decouples the central logic from both ends, becoming a sort of "middleware" server that connects request handlers to external services, with neither end requiring knowledge of the other.

Crosstalk itself is a core server program with well-documented APIs for its extensions:

- An unrestricted number of arbitrary interfaces to external services, such as Instant Messenger networks and HTTP
- An equally unrestricted number of "back-end" request handlers that can connect to web services, use other programs, or generate responses themselves
- Modules to route requests from external interfaces to the appropriate back-end request handlers
- User-defined system management and monitoring interfaces.

This document describes

- the functions of, and interfaces for, writing Crosstalk modules
- Crosstalk's internal communication system between modules
- Other miscellaneous features of the system
- Technical issues concerning Crosstalk's implementation
- Descriptions of the example modules developed for Crosstalk

Document Conventions

In this document, the following text formats have special meanings:

- *Italicized Helvetica*
  This string of text is the name of a file. It may contain interpolated environment variables.
- `Courier`
  Represents code: function or variable names, subroutine signatures, or other terms.

Modules

Crosstalk is built around the idea of modules, independently-executing units of code that perform specific tasks and communicate via messages. The entire Crosstalk system is composed of modules that operate without direct interaction, communicating with each other and the main program (which can itself be considered a module). All modules conform to specific conventions and APIs that allow them to operate in the Crosstalk server without conflicts.

Module File Conventions

All module files must be syntactically correct Perl, following these conventions:

- The name of the file is the name of the module. Thus, no `.pl` extension should be used. The module must be placed in the proper directory within the Crosstalk home. If you have a Service module called HTTP, it must reside in the file *$CROSSTALK_DIR/Services/HTTP* (see the section on Crosstalk Installation).
- Modules should not put themselves into a namespace the way Perl libraries do.
- Since modules will be loaded with the perl function `do`, they must return a true value, just like regular Perl libraries. This is usually accomplished by having a true statement such as `1;` on the last line of the file.
- All modules must define the functions `Init` and `Quit`. These function will be called by the Crosstalk bootstraps when the module is loaded or unloaded. These functions can be used to perform operations such as opening and closing files or sockets, or initializing hashes and arrays. They must be defined even if the module has no work to do in them.
- Other functions must also be defined, depending on the module types. Those functions will be described along with the module types themselves.

All modules are loaded in the protected namespace `Local::`, so they may define whatever other functions or variables they want without worrying about name collisions. The following names are exported into the Local namespace, so that the modules may use Crosstalk resources without specifying the `main::` package:

- `&Receive`
- `&Send`
- `&Log`
- `&SetTimer`
- `&ConfigGet`
- `&ConfigSet`
- `&ConfigDelete`
- `$name`
- `$type`
- `$basedir`

The last three are variables: the name and type of the module itself, and the location of the root Crosstalk directory (equivalent to the environment variable *$CROSSTALK_DIR*). The functions will be described later on.

Module Configuration

Crosstalk supports a system for storing and retrieving configuration information for modules. Configuration data is stored in the *$CROSSTALK_DIR/Config* directory, under subdirectories for each module type.

Only one hash is stored for each module; it is written out using the standard Perl `Data::Dumper` module, and read in during module initialization. This file looks just like Perl code, and can be edited by hand to set up a module. Hash keys and values can be manipulated with the following functions:

`$value = ConfigGet($key)`

> Returns the configuration value associated with the specified key. The value is not necessarily a scalar, and in fact can be any arbitrarily complex Perl data structure (barring non-serializable objects like functions and typeglobs).

`ConfigSet($key, $value)`

> Sets a configuration value, which may be a reference to a data structure. If the key does not exist, it is created. The configuration file is saved immediately.

`ConfigDelete($key)`

> Deletes a configuration key. Does not fail if the key does not exist. The configuration file is saved immediately.

Loading Modules

To make the interface for modules as simple as possible, much bootstrapping work is done in several functions within the Crosstalk executable:

`($success, $resultString) = InitModule($moduleType, $moduleName, @arguments)`

> This function is called by the Main thread whenever a module is to be loaded. `InitModule` creates the necessary data structures (message queues, etc) for the module, starts the actual thread, checks for the module's successful initialization, and reports the success of the loading back to its caller.

`LoadModule($moduleType, $moduleName)`

> This function acts as the root of the thread that is created by `InitModule`. `LoadModule` does the actual work of parsing and loading the module file, checking for errors, loading the module's *Config* file, running the Init function of the module, and setting up the protected namespace for the module's declarations.
>
> `LoadModule` reports its status back to `InitModule` via semaphores, since it cannot use `return` without terminating the thread. After successfully loading the module, its runs the appropriate `Run` function (described below) inside an `eval` block. This block acts as a try/catch to trap errors that may occur during execution of the `Run` function. If an error is caught, `LoadModule` logs it, sends Main a command to restart the failed module, and exits.

`RunMonitor, RunHandler, RunService, RunRouter, RunManager`

> The actual running of a module is accomplished by one of the five functions listed above. Depending on the type of module, these functions may do nothing more complicated than simply call the `Run` function defined in the module's file, or it may do the work of receiving messages and calling appropriate functions within

the module. The exact structure of each kind of `Run` function is described below, in the module type descriptions.

Module Types

Main

The Main module can be considered the "root" of the program; it is the initial thread from which all other threads are spun off. The Main thread takes care of loading and unloading modules, handling errors, and executing commands from the user. Because it is the initial thread in the program, it has no bootstrapping routines.

Monitor

The Monitor thread provides a flexible interface for recording information about the state of the running Crosstalk program. During program execution, modules can call the function

```
Log($eventType, $data, $isError)
```

to send an event to the active Monitor module to be logged. The `$eventType` argument allows the Monitor to filter events by their general category. There are no set values for `$eventType`, but common ones include Error, Initialization, Routing, Handling, and Message. `$isError` allows for explicit marking of an exceptional event.

The logged events are placed in a queue, which is continually polled by the Monitor bootstrap function. When an event is received, the bootstrap function calls the module-defined function Record with the following arguments:

```
Record($moduleType, $moduleName, $time, $threadId,
$filename, $line, $subroutine, $kind, $string,
$isError)
```

This function is the only definition required in a Monitor module, besides `Init` and `Quit`. `Record` gives the Monitor module comprehensive information about when and where the event occurred. The module may then do as it wishes with the event. The implemented Monitor module, File, simply records all events to a log file (the desired event types are specified in the configuration file). Other Monitor modules could post updates to a web page, or send out email or instant messages about exceptional events like errors.

Only one Monitor module is active during the running of Crosstalk, and it can not be dynamically loaded and unloaded like the other modules.

The `RunModule` function inside *Server.pl* takes log entries (which are three-tuple arrays, not `Crosstalk::Message` objects) off its queue directly, without using `Receive()` (see the discussion of the Crosstalk communication system below). It calls `Local::Record` for each log entry. If the object coming off the queue is a

`Crosstalk::Message::Command` of '`Quit`' from the Main module, `RunMonitor` calls `Local::Quit`, then exits the thread.

Service

Services are Crosstalk's interfaces to the outside world: they are responsible for receiving requests from the user and sending responses back. Services are typically synchronous, such as HTTP/SOAP, Instant Messenger, or command-line based, but they may also be asynchronous, such as email.

Due to the fact that Service modules must listen to the outside world, their interface is much simpler, requiring more work and customization inside the module itself. The only function a Service module must define is `Run`, which is called by the `RunService` function inside *Server.pl* (and is the only thing that `RunService` does).

This function, `Run`, is then responsible for connecting to any outside sockets and listening for responses. However, the Service module must also be listening to interior messages from Crosstalk itself: commands to shut down or suspend the generation of requests. There are two main methods to accomplish this: spin off another thread to wait on `Receive()` for internal messages, or to use non-blocking methods to continually switch between looking at the internal queue and the external socket. The former would be more ideal, but Perl's thread system introduces its own problems, not the least of which is the inability to share filehandles between threads, destroying much of a multithreaded Service's utility.

In practice, then, a Service will still have two threads: one that does request-at-a-time listening to the outside world, and another that waits for internal messages to shut down or suspend listening. In that case, the inward-facing thread can send a special, escaped message to the outward-facing thread's normal listening channel, telling it what to do. This design is inelegant, but the best compromise given the limits of Perl's thread system.

Routers

Routers are responsible for examining the messages that come from Services and determining which Handlers they are bound for. Routers are given rules for each Handler module, and must apply those rules to incoming messages, being wary both of no matches and of multiple matches.

Router files must define three main functions:

`$success = AddRules($handlerName, @ruleset)`

Record the given rules for the specified handler. The ruleset may be rejected if one or more rules are invalid.

`$success = RemoveRules($handlerName)`

The inverse of `AddRules`. This is usually called when a Handler is being unloaded.

`$request = RouteRequest($request)`

Check the given request against the current rulesets, mark it appropriately (see the discussion on communication systems below), and return it.

The actual format of rules is completely specified by the Router, although it is required that each ruleset for a Handler be a list of simple strings. These strings could be individual characters, keywords, regular expressions, or more complex grammar types. The Router may reject rulesets that are not valid (i.e. a regular expression with syntax errors).

Routers may perform pre-processing on requests in the course of routing them, and storing results in the request using the `additional_info` method. An example is the Regex router, which pulls out groupings from the request text and places them in `additional_info` so that the Handler does not have to repeat the work.

Routers are given priorities in Main's config file. This determines which Router sees a request first. If that Router cannot find a match, it moves to the next Router in the list. If not Router can match a request, or a Router finds a request to be ambiguous, it is returned with the appropriate status to the originating Service module.

The RunRouter function inside *Server.pl* checks both for incoming requests and for commands from Main to add/remove rules and to quit, and calls the appropriate functions.

Handlers

Handlers perform the actual information-gathering and request processing. After a request has successfully been routed, it is processed in the destination Handler by calling the function `HandleRequest` with the request object as its sole argument. `HandleRequest` should do whatever is necessary to fulfill the request, set its response text and success status, then return. `HandleRequest` may mark a request as unsuccessful if it is nonsensical or impossible.

When `HandleRequest` receives a request object, it has been marked with which Router matched it, as well as the results of any preprocessing the Router has done. This allows the Handler to quickly decide what to do with the message, including how to interpret the contents of the request text, instead of forcing it to essentially duplicate the Router's effort.

A Handler module is also required to define the hash `%rules`, which should contain the names of Routers and the rulesets for them. Upon loading, `RunHandler` will pull out these rules and send commands to the active Router modules to add them.

If a Handler dies during the processing of a request, this could hang the Service waiting on that request to return. Instead, `RunHandler` calls `HandleRequest` inside its own `eval` block. If there is an error, it zeros the request's `success` flag and sets the error string, and sends it back to the Service. The exception is then rethrown so that `LoadModule` may process it normally.

Managers

Managers are the interfaces for Crosstalk administrators. They allow run-time inspection of the current state of the Crosstalk system, as well as loading and unloading of Service, Handler, and Router modules. Only one Manager can be loaded, and that module must be specified during in the Main configuration file.

Managers need only define one function, `Run`, in their files. This single function is called by the `RunManager` bootstrap. `Run` is responsible for listening for commands from the administrator, then sending the appropriate `Crosstalk::Message::Command` messages to the Main module, which actually does the work of executing the command.

Interfaces that Managers can use may include email, HTTP, or command-line programs. Specifics, including security measures, are left strictly up to the individual Manager modules.

Communication System

The use of threads to isolate modules requires a sophisticated communication system. Like many multi-threaded systems, Crosstalk uses message passing to take requests from the Service interfaces, through the Routers, to the Handlers, and back out to the Services.

Message Classes

With the mechanism for passing messages established, the actual messages themselves are rather simple. Messages in Crosstalk are members of a simple hierarchy of classes that encapsulate message origin, destination, other parameters, as well as allowing for arbitrary data to be passed back and forth. For more extensive description of the interfaces presented here, see the Crosstalk POD documentation.

Crosstalk::Message

The base class Crosstalk::Message provides methods for identifying messages, tagging them with success and error conditions, and setting destinations.

The origin module type and name of a Message object is set automatically in the constructor, through use of the localized `$type` and `$name` global variables. The destination may be specified at creation, or deferred until the destination is determined (for example, until a Router module correctly routes a request).

All messages have both success and error status. The `success` attribute is initially undefined, and should be set to either 0 or 1 once a message has been processed as far as possible. If the message is unsuccessful, the attribute `error` should be set with the error string.

Each Message created has a unique number stored in the attribute `id`, which can be used to keep track of outstanding messages. The Main module uses this attribute to determine what commands to various modules have not yet been completed.

Crosstalk::Message should never be instantiated itself; in fact, the class' constructor checks for this case and dies if it is attempted. It is intended as a virtual base class for the specialized message types.

Crosstalk::Message::Command

Crosstalk::Message::Command is used for internal, inter-module messages: "Stop your thread's execution", "Load the routing rules for this Handler", or "Tell me about your current state". Command messages typically originate in a Manager module, which takes administrator input, converts that into a Command, then sends the command to the Main thread. The Main thread handles the command as appropriate, which usually involves creating another Command message to send to the module whose management is being requested.

A list of arguments and results may be attached to a Command message. For example, the arguments are used to send a Handler's ruleset to a Router in an "AddRules" command. The results list is typically used in commands that poll for the current state of the system or of individual modules.

Command messages may be flagged as requiring a response. Usually, a response to the command is desired, either to return information (as in a poll of a module's current state) or to indicate success or failure of the command's execution. In some cases, however, a command is not even returnable. Take the example of a module whose bootstrap function catches a runtime exception. All modules run their functions inside the equivalent of try/catch blocks, which in Perl equates to `eval { … }; if $@ { … }`. If an error is caught in this manner, the module has no choice but to shut down and terminate the thread. However, the Main module must be informed of the module's failure, and told to restart it. The logical way to do this is by sending Main a Command message indicating such. But, if Main tried to return the command, it would be sending it back to a thread that has finished! This is an exceptional case, but its nature requires the use of non-mandatory responses to commands.

Crosstalk::Message::Timed

Crosstalk::Message::Timed is a specialty class, designed for the asynchronous communication of the Timed Services Interface (which is described later). Timed messages are only of use to Handlers, which may be designed to set up periodic "pushes" of data out to the user. These messages have no origin; instead, they are provided only as a notification that an event should occur.

A Timed message contains as its argument a single string, which is set during the creation of the timer.

Messaging Implementation

The actual message-passing apparatus involves nothing more than instances of the standard Perl class Thread::Queue. This queue class implements the locking and sharing needed to pass simple scalars from one thread to another. Each module is allocated one queue upon initialization, and it can pull messages out of this queue as necessary. The actual queues are hidden by two functions:

```
Send($message)
$message = Receive().
```

`Send` places takes a message and routes it to the appropriate module's queue; `Receive` waits for a message to come in to the current module's queue and returns it.

`Receive` is a blocking operation; it will wait until a message is placed into the queue by another module. In almost every case, this is desirable behavior, and helps to simplify the communication logic.

A very important feature of `Receive` is that it automatically determines which module is currently active, and so on which queue to wait. This is accomplished via Perl's dynamic scoping operator `local`; during initialization, `LoadModule` localizes the variables `$type` and `$name` to be the module type and name of the current module, making them in effect global variables that have values specific to each thread. Numerous Crosstalk functions use these localized variables to ensure that they are operating on the correct module's data.

Message Routing

Message routing is largely automatic in Crosstalk; a module does not have to specify where a message should go, and the "destination" fields of a message are not always where the message will go. Instead, `Send` routes messages according to the following rules:

```
If the message is a Request:
        If it has no destination set and has not visited all Routers:
                Its current Router number is incremented (starting at 0), and it is sent to
                that Router, as determined by the router priorities.
        If it has no destination set and it has visited all the Routers:
                No Router matched it, so it is marked as unsuccessful and sent back to the
                origin (usually a Service)
        If it has a destination set and has not had its success marked:
                It is sent to the destination
        If it has had its success marked:
                Send back to the origin, for either successful or unsuccessful requests.
                Unsuccessful requests can include ambiguous requests from a Router or
                failure from a Handler.
If the message is a Command:
        If the message has not had its success marked:
```

```
        Send to the destination.
    If the message has had its success marked and is returnable:
        Return to the origin.
    If the message has had its success marked and is not returnable:
        Drop.
If the message is Timed:
    It is sent directly to the destination module.
```

In this manner, all possible types of messages are routed automatically, without the individual modules having to do any routing decisions.

Other Features and Notes

Timed Events

While Crosstalk is designed to support mostly synchronous communications – a user submits a request, it is handled, and the response is returned – there is also usefulness in allowing for asynchronous events. A canonical example would be a Handler that wants to watch an RSS news file, and send a user an instant message whenever new stories appear on a website.

To support this, Crosstalk has a simple Timed Services interface that allows for modules to set messages to be delivered back to them at specified times in the future.

Interface

`SetTimer($time, $message)`

> Calling the function registers the text string `$message` to be returned to the calling module at time `$time`. `$time` is a Unix-style timestamp, the number of seconds after some date, such as midnight, January 1, 1970 GMT.

The Perl built-in function `time` returns the current timestamp. Because of the platform-specific nature of timestamps, as well as allowances for timezones, timestamps should always be calculated by taking `time` and adding the desired number of seconds.

When the timer goes off, a `Crosstalk::Message::Timed` message is sent to the module's message queue, where it may be retrieved with `Receive`. The simple text string `$message` will be held in the Timed object.

Recurring timers cannot be set with this method. If desired, the calling module should call `SetTimer` again once the Timed message has been received and processed.

Implementation

The official advice on using signals (which are what all timers require) and threads in Perl is "don't". Although much work with threads has been done, a large issue remains in that Perl has no way of guaranteeing which thread will be active when the signal is received; timers cannot be directed towards specific threads or objects, as they can in languages like Python and Java.

This would seem to forbid asynchronous events, but the communications system of Crosstalk actually makes it relatively simple. Upon startup, Crosstalk sets a timer to go off every second, using the `itimer` system call as exposed in the `Time::HiRes` module. When a module calls `SetTimer`, the timed message is stored in a shared hash with its time of execution (in seconds) as the key. Every second, the timer wakes up a `SIGALRM` handler, which looks in the hash for messages that have timestamps less than the current time. Those messages are removed from the hash, wrapped into Timed objects, and placed in the queue of the destination module using `Send`.

This solves several major problems. For one, no additional mechanism of handling timed events is necessary, since the modules will get the message through `Receive` as it would anything else. Second, it doesn't matter what module is currently running when the timer goes off, since the hash of messages is shared between all threads, and the message is delivered to the thread's message queue.

One disadvantage is that, because of the 1-second interval of the timer, as well as the unpredictability of thread scheduling, fine-grained events are not possible. For Crosstalk's purposes, this is not problematic, as most users will not be able to notice a one- or two-second delay of, for example, their RSS feeds.

Security Issues

If a system such as Crosstalk is to be anything more than a toy, security inevitably becomes a concern. Crosstalk is built to allow an arbitrary number of users to access arbitrary services from arbitrary interfaces, and a Crosstalk system wishing to provide useful services will need to make sure that those services are not misused or abused.

Crosstalk allows a basic framework for security, by attaching a username to each incoming request. This username logically maps to the kind of Service producing the request: an email address, an Instant Messenger screenname, or an authenticated HTTP username. Handler modules may then examine the Service and username combination and decide what kind of requests may be honored. Alternately, the username can allow Handlers to have a memory of "sessions", being able to recognize a chain of requests coming from one source.

This is a positive feature, since it allows all of the work of setting up user accounts and authenticating to be offloaded to the Service. With usernames and domains (Services) effectively provided, it is up to the individual Handlers to implement their own security policies. Although this may seem like an undue burden upon Handler writers, building a comprehensive, flexible security into Crosstalk's core would have been too difficult. Handler that manage stock portfolios will want very rigorous security; an Eliza program may want none. Besides, with authentication and domains already handled, all that remains is writing heuristics for resource access. Crosstalk provides enough of a platform that very sophisticated security policies may be built on top of it with little effort. See the entry for the Shell Handler for an example.

Crosstalk Installation

Requirements

The following environment must be in place to use Crosstalk:

- A Unix system is required. Only Mac OS X has been tested, although Linux and *BSD should be compatible.
- Perl 5.8.0 must be installed, and specially compiled to use ithreads.
- The Perl package Time::HiRes must be installed from CPAN

Crosstalk may be easily installed by simply decompressing the tar-gzip file, editing the config files inside to the desired state, and setting the `CROSSTALK_DIR` variable.

Installation Layout

Crosstalk uses the following directory structure to store its files:

$CROSSTALK_DIR/	The root directory of Crosstalk. This directory must be defined as the environment variable `CROSSTALK_DIR`
Config/	Contains all the configuration files for modules. Each module type has its own subdirectory
Handler/	
Main/	
Main	The Main/ configuration directory only contains a single configuration file, for the Main thread itself. This configuration, unlike all the others, stores more than one variable and has predetermined fields.
Manager/	
Monitor/	
Router/	
Service/	
Documentation/	Contains all the documentation, in POD format
HTML/	HTML files created by pod2html are stored here.
Handlers/	Contains the files for Handler modules
Library/	This library directory is included with the use lib directive, and holds all of Crosstalk's internal libraries and classes
Crosstalk/	This second directory is necessary to be able to prepend module names with Crosstalk::
Config.pm	This library is not a proper module, but merely defines functions for reading and saving configuration information for modules

Encode.pm	Holds routines for "encoding" objects so that they may properly be passed between threads.
Message/	
Command.pm	The Command message subclass
Request.pm	The Request message subclass
Timed.pm	The Timed message subclass
Messsage.pm	The base class of Crosstalk inter-module messages
Timer.pm	Defines the functions for setting timed events and handling the timer when it goes off
Managers/	Contains the files for Manager modules
Monitors/	Contains the files for Monitor modules
Routers/	Contains the files for Router modules
Scripts/	Holds scripts useful to running the Crosstalk system, including commands to start/stop the server and update the HTML documentation
crosstalk	Controls execution of the Crosstalk system. Takes one of three arguments: start, stop, or restart.
makedoc	Uses the pod2html program to convert all the POD documentation in the *Library/* and *Documentation/* directories to generate HTML in *Documentation/HTML/*.
send-request	Used for sending a request message to the HTTP Service
start_crosstalk	A bootstrap script that properly runs *Server.pl* in the background. Should not be executed directly; instead, use "`crosstalk start`"
Server.pl	The actual Crosstalk executable
Services/	Contains the files for Service modules

Technical Issues

Use of Threads

Crosstalk is built as a multithreaded application, which is something of a rarity in the Perl world. It is only within the last two years that Perl has had thread support, and only within the last nine months that those threads have actually been usable. The decision to use threads came as a tradeoff with heavy penalties.

Disadvantages

A primary disadvantage of using threads in Perl is that it requires a custom-built perl executable of the latest version, 5.8.0. All dynamic libraries that link to external compiled code must also be rebuilt for threads. This new, thread-enabled build of Perl not only runs slower and consumes more memory, even on non-threaded programs, it also is slightly less stable.

Besides the problems of the Perl executable, almost no Perl libraries have been rewritten to be thread-enabled. If an object from a Perl library is only to be used in a single thread, this is not a problem, since be default all data is duplicated between threads. However, if an object is to be passed between threads, that object must be explicitly marked as shared in the library's constructor function, as must any objects it contains, etc.

Although this problem can be solved by editing every library to explicitly make object instances shared, it is impossible for modules that are based on filehandles. Filehandles in Perl 5 are not first-class objects like scalars or hashes; instead, they are references to unique, autogenerated typeglobs. Typeglobs cannot be shared between threads, for a very good reason: while moving a shared scalar or reference into another's thread's address space presents no problems, moving a typeglob has the potentially disastrous effect of modifying that thread's symbol table (symbol tables are always unique to threads in Perl). In the worst case, a filehandle being imported from another thread could overwrite a filehandle from the same package, since the typeglob names are usually produced in predictable manners (something like `"prefix" . $i++`).

This restriction, which is quite reasonable, prevents threads from giving filehandles to each other – including socket connections. The original design of the HTTP service had a single listener thread that would create a connection object, then hand it off to a worker thread that would process the request. This is impossible, since the filehandle cannot be shared, and as a result, the HTTP service only handles one connection at a time. Although it is of little comfort now, Perl 6 will fix this problem by making filehandles first-class entities in the language, and allow them to be shared between threads.

Another complication is that programming with threads is that, since the Perl debugger is not thread-enabled, getting information about the state of a running program is limited to print statements or whatever output is sent through the Monitor module.

If that were not enough, Perl's thread model is extremely heavyweight. In creating a thread, Perl actually creates an entirely new interpreter process in the same address space, duplicating all the data currently addressed, except for what has been explicitly marked as shared using the "`my $scalar : shared`" construction. As such, creating threads "on the fly" to handle individual requests would have unacceptable performance. This hit can be mitigated somewhat, however, by the standard techniques of either creating and maintaining a constant pool of threads, or, in the case of heterogeneous threads, creating them all at startup, to move the cost away from run-time.

Advantages

With such problems arising from threads, their benefits would have to be very great to choose their use. Despite their numerous problems, threads offer several indispensable benefits.

The main reason to use threads is to simplify the communication model between modules, and to allow for pieces of the program to execute asynchronously. While a non-threaded Crosstalk could conceivably use the same module interfaces, but only handle one request at a time – in fact, early versions of Crosstalk did just this – it would make it impossible to have asynchronous events or simultaneous handling of requests, significantly decreasing the server's robustness and utility.

Threads also make the writing of modules with external interfaces much easier. One of Crosstalk's design parameters is that it support an arbitrary number of interfaces to the outside world. Typically, these interfaces involve opening up a socket for connections, then listening for a request. The only way to accomplish this in Unix without multiple processes (which introduces its own communication problems) or threads is by use of `select`. This is a perfectly reasonable way to write a web server, or any kind of process that handles a number of homogeneous connections. However, Crosstalk supports any kind of protocol for incoming data, making the logic of `select` much more difficult and destroying the componentized nature of Service modules.

Although these may seem like small design gains, they are essential to creating the flexibility and simplicity of the Crosstalk module interfaces. By using threads, Crosstalk subsumes all of the complexity into the server core, allowing the modules to be written easily and without concern for the rest of the system.

Local Scoping

Many of Crosstalk's utility functions, including the messaging, configuration, and timing subsystems, need to know which module is calling their functions. If implemented in the most obvious way, this would require every function call to have arguments specifying the type and name of the module. This is both inconvenient, as it makes the interfaces needlessly complicated for the programmer, and dangerous, since it requires the module's name and type to be specified in numerous places, increasing the odds that one of them will be incorrect.

To solve this problem, Crosstalk exploits an ancient, normally useless feature of Perl – localized variable scoping – and a normally undesirable effect of threading – the duplication of all data.

Localization allows a lexical scope – a function, or a loop – to take a package variable and temporarily give it a new value. The old value is restored upon the scope's exit. This is a fine workaround if lexical variables are not available, but has one significant side effect: functions called by the localizing scope also get the localized value of the variable.

This allows *Server.pl* to set up two variables – `$main::type` and `$main::name` – then, in `LoadModule`, localize them to the type and name of the module being initialized. The variables are aliased into the module's protected namespace, and instantly, any function invoked by a module – such as `SetTimer` or `ConfigGet` – knows which module has called it.

Although the `main::` package is being modified, each thread gets its own copy of the package variables, except those explicitly marked as shared. This allows `$type` and `$name` to be set and localized without interfering with the other threads.

Example Modules

This section describes the example modules developed along with Crosstalk, demonstrating parts of its design and functionality.

Monitors

File

This simple Monitor writes out all events it receives to a log file. It can be configured to screen log messages based on their event type, to let Crosstalk administrators only record desired messages.

Services

HTTP

The HTTP Service uses the Perl HTTP::Daemon library to listen on a user-specified port for incoming POST requests. It does not perform CGI decoding, but rather takes the POST contents as the plain-text request. The response text is then returned as the HTTP response (again, without CGI or HTML encoding).

HTTP does not perform authentication (although it could); instead, every request it generates has a username of 'Anonymous'.

AIM

This Service signs onto the AOL Instant Messenger network and listens for incoming messages. Each message is translated to a request, and the username of the request is set to the screenname of the message's sender. The response text is sent back to that screenname as an Instant Message.

Routers

Keyword

This simple Router takes lists of words as its rulesets. Incoming messages are tokenized on whitespace and punctuation, and compared against each set of keywords to see how many match. The Handler whose ruleset has the most matching keywords is set as the destination. Ties are not marked as "Ambiguous", but are rather broken in first-come-first-served order.

Regex

The Regex Router applies Perl 5 regular expressions against incoming requests, and routes the request to the Handler who has a match. Ambiguous queries are marked as such, and sent back to the originating Service.

Regex also makes use of the `additional_info` attribute of Request objects. If a pattern in a ruleset includes capturing parenthesis operators, those captured subpatterns are returned in the `additional_info` list, preceded by the index of which pattern matched. The Handler receiving the request then knows what form the request text follows, as well as have the significant parts already isolated. This eliminates the duplication of effort that otherwise would occur when the Handler reparsed the request text. See the description of the Eliza handler below for an example of this.

Managers

Web

The Web Manager provides a simple interface to query the current state of the system, as well as load and unload modules. It also uses the HTTP::Daemon library to bind to a user-specified port and present three services (accessed as HTTP GET paths):

`/status`	Presents a page that lists the currently installed modules, as well as all available modules, with links to add or remove as appropriate
`/add`	When accessed with CGI arguments of `name` and `type`, loads that module on the fly. The `/status` page is then reloaded to reflect changes.
`/remove`	Like `/add`, uses `name` and `type` CGI arguments to dynamically unload a module, then reload the `/status` page.

The Web Manager does not perform any security checks, but could be altered to include HTTP authentication by using a different Perl library.

Handlers

eliza

Implements a standard Eliza program with a limited set of patterns. The Eliza handler uses the `additional_info` attribute of its Request object to form response text without complex logic. Each rule (or regex pattern) in the ruleset for the Regex Router has a corresponding anonymous subroutine in an array called `@responses`. When a request is received, the first element of `additional_info`, the index of the matched rule, is used to call the appropriate subroutine, which returns a response string from interpolating the matched subpatterns of the original request – the rest of the values in `additional_info`. New patterns and phrases can be added simply by inserting a new pattern and response subroutine at the appropriate places in the two arrays, without any additional logic.

verbose_stock

Fetches stock information from Yahoo! Finance and returns it. Upgraded from the original terse_stock handler to be more functional, as well as use the Regex Router. Plain-English requests like “Tell me the close price of msft” are supported, and the response will also be given in English.

timer_test

A demonstration of the Timed Services interface, the timer_test Handler simply registers a message to be returned to it 10 seconds after loading. When that message is received, it prints “Hello, world!” to stdout.

Shell

The Shell Handler both allows remote machine access and implements a security policy on top of the Crosstalk username/domain framework.

This module allows a user to send a command in the request text, which will then by executed as a command-line program using Perl’s backtick operator, and the output returned as the response text. To prevent the glaring security flaw of giving anyone shell access to the Crosstalk system’s machine, the Shell handler only permits certain users to execute commands. In its configuration file, lists of usernames are associated against Service names, and those lists associated against the usernames that Crosstalk might be running under on its own system. When a request comes to the Shell handler, it looks at Crosstalk’s current username, the name of the incoming request’s Service, and the name of the user on that Service, and sees if that user is allowed to run commands. This is a very simple security policy, but it provides very tight security, barring only the compromise of the user’s domain and name (for example, a stolen Instant Messenger account).