DAVE JOSEPHSEN

iVoyeur: *you* should write an NEB module.

David Josephsen is the author of *Building a Monitoring Infrastructure with Nagios* (Prentice Hall PTR, 2007) and Senior Systems Engineer at DBG, Inc., where he maintains a gaggle of geographically dispersed server farms. He won LISA '04's Best Paper award for his co-authored work on spam mitigation, and he donates his spare time to the SourceMage GNU Linux Project.

dave-usenix@skeptech.org

The Nagios source code can be downloaded from http://www.usenix.org/publications/login/2008-10/nagfs.c.

FOUR YEARS AGO, I ATTENDED THE Nagios BoF at LISA '04 in San Diego. It was being thrown by a few employees from Groundwork, including Taylor Donditch, the author of fruity. Despite the fact that the BoF was a day before the tech sessions started and therefore not on the official BoF schedule, it was standing room only. For me this was an amazing contrast from 2001, where I mentioned Nagios in a network monitoring BoF and was met by blank stares.

In 2004, Nagios 2 was a fairly new beast, and Taylor was excitedly waxing prolific about the Event Broker interface. That night, all questions to Taylor led back to the Nagios Event Broker. Improved passive checks? NEB. Scalability problems? NEB. Goldfish dead? NEB. This was for good reason: The NEB put a lot of power in the hands of the sysadmin and promised to eliminate or at least reduce the myriad influx of Nagios-related Perl kludges on Nagios Exchange. If you had a problem with Nagios, there was now a correct way to fix it, and that was to write an NEB module. There was no doubt in my mind that everyone in that room would hurry off straightaway and create all sorts of interesting and useful event broker modules. I knew that by morning a wiki would have appeared somewhere, with 30 or so of them a template for making your own and a Web comic making fun of people who used them. For my own part, inspired, I immediately dove into the NEB headers (well, three or so days later, once I sobered up).

Four years later, the Perl kludges have only grown in number, whereas the NEB modules are nowhere to be found. I find this surprising and unfortunate, because the NEB is an elegant solution, and it has the potential to help far more sysadmins per line of code than any script you'll likely find on Nagios Exchange today. For once, the folks who wrote the application recognized our need to customize their program and actually engineered their app in such a way that it can be fairly easily modified to suit our needs. Further, the mechanism they've created is as portable as the app itself, and it could easily do for Nagios what the distros did for Linux. Today, I can internally modify Nagios to customize for a particular problem domain and distribute my customized Nagios in the form of a small piece of shared-object code that can be switched on or off by anyone who uses Nagios. That's cool.

So since Rik tells me this issue will be available at LISA, I'd like to honor the 2004 Nagios BoF by taking some time to explain how the NEB works and hopefully inspiring some of you to sober up and scratch some of your Nagios itches by writing NEB modules. First, I'll give a short description of the architecture, and then we'll walk through a working NEB module I wrote called nagfs, which implements a filesystem interface to Nagios. Since I'll be using my own code as an example, I'll be stuck talking about Nagios 2.x in this article. That's a bit of a bummer because several very empowering changes have been made to the architecture in 3.x; perhaps I'll cover those in a follow-up next time.

The Event Broker itself is a software layer between Nagios and small, user-written shared object files called event broker modules. The Event Broker initializes all of the modules when Nagios first starts, so it knows what events the modules are interested in. Then it sits around waiting for interesting events to occur, passing out memory handles for each interesting event to the module that is interested.

NEB modules are shared libraries written in either C or C++. The NEB module registers for the types of events it is interested in and provides function pointers to functions that presumably do things with the events they receive. Each NEB module is required to have an entry and exit function and, beyond that, can do pretty much anything it wants. The interesting thing about this architecture is that Nagios globally scopes just about everything (by design), so from the perspective of the NEB module, the sky is the limit.

That is to say, because pretty much all the interesting functions and structs are globally scoped, as long as Nagios's execution pointer is in the module's address space the module has the power to change anything it wants. It can, for example, insert and remove events from the scheduling queue or turn on or off notifications or do things such as preempt given check commands or postprocess returned data from service checks. In a nutshell, anything that can be changed at runtime can be changed by the module. Strictly speaking, the module need not even register to receive events; upon initialization the module could schedule its own call-back routines in a timed fashion and do its job using nothing other than Nagios's scheduling engine. It could, for example, wake up every morning and change the value of the day-pager's email address, or wake up every 5 seconds and provide state information to a visualization front-end.

So what sorts of events can a module subscribe to? In Nagios 2.3.1, the version of Nagios I'm using as I write this, there are 31 total call-back types, although some of them are reserved for future use. These constants are defined in nebcallbacks.h, in the "includes" directory of the tarball. Listing 1, on the next page, contains some of the call-back type constants.

The available callbacks cover every type of event that can happen in Nagios. An NEB module may register to receive information about any or all of these event types. Once it initializes all the modules, the Event Broker waits for events matching the type subscribed to by the module and, upon receiving one, gives the module information about the event, as well as a handle to the relevant data structures.

For example, if the module registered for EXTERNAL_COMMAND_DATA, the Event Broker would notify it every time an external command was inserted into the command file. A handle to a struct that defined the command would accompany the notification. The module could inspect and optionally change any of the information in the command struct or even delete it altogether. But enough talk about the architecture; the best way to learn about the NEB is to see how these modules work in practice.

```
NEBCALLBACK_FLAPPING_DATA
NEBCALLBACK_PROGRAM_STATUS_DATA
NEBCALLBACK_HOST_STATUS_DATA
NEBCALLBACK_PROCESS_DATA
NEBCALLBACK_TIMED_EVENT_DATA
NEBCALLBACK_LOG_DATA
NEBCALLBACK_SYSTEM_COMMAND_DATA
NEBCALLBACK_EVENT_HANDLER_DATA
NEBCALLBACK_NOTIFICATION_DATA
NEBCALLBACK_SERVICE_CHECK_DATA
NEBCALLBACK_HOST_CHECK_DATA
NEBCALLBACK_COMMENT_DATA
NEBCALLBACK_SERVICE_STATUS_DATA
NEBCALLBACK_ADAPTIVE_PROGRAM_DATA
NEBCALLBACK_ADAPTIVE_HOST_DATA
NEBCALLBACK_ADAPTIVE_SERVICE_DATA
NEBCALLBACK_EXTERNAL_COMMAND_DATA
NEBCALLBACK_CONTACT_NOTIFICATION_DATA
NEBCALLBACK_ACKNOWLEDGEMENT_DATA
NEBCALLBACK_STATE_CHANGE_DATA
```

LISTING 1: SOME NEB CALLBACK TYPES

Nagfs is a filesystem interface that represents the state of a running Nagios daemon. Each host monitored by Nagios has a directory in the filesystem, and each service on that host has a file. The contents of the file match the Nagios service state for that service. For example, if the httpd service on box1 was down, then /usr/share/nagios/status/local/box1/httpd would contain a '2'. Most people scrape HTML from the Web interface to get this kind of info, so you can imagine how handy it is to just be able to do `grep -rl 2 /usr/share/nagios/status/local` to find all the services in a critical state instead. Nagfs keeps the filesystem up to date by subscribing to state change events. Every time a service changes state, the event broker tells nagfs, and nagfs updates the filesystem immediately. No waiting for an external event_handler to fire; if Nagios knows about it, so does nagfs.

The complete source code for nagfs is a bit long to print here. If you'd like to compile it yourself, grab a copy of the source off my blog (www.skeptech.org/?p=35), from http://www.usenix.org/publications/login/2008-10/nagfs.c, or from NagiosExchange along with the Nagios source code from nagios.org, and follow the instructions therein. What we can do is examine some key portions of the source. Let's start with the function declarations:

```
/* Nagfs Functions */
void nagfs_reminder_message(char *);
int nagfs_handle_data(int,void *);
int nagfs_write_service_status(char *, char *, int, int);
int nagfs_write_host_status(char *, int, int);
int nagfs_check_for_softfiles(char *);
```

An event broker module is only required to have two functions, `nebmodule_init` and `nebmodule_deinit`. The function `init` gets called when our module is first initialized, and `deinit` gets called when Nagios quits and we get unloaded. The functions I've declared above are all optional, and I mostly declare them up front so that the program follows a more linear progression and is therefore easier to write about. The basic strategy is that our `init` function will register for event callbacks and will call `nagfs_handle_data` to handle the data we receive from the broker; `nagfs_handle_data` will in turn call the other functions as needed to update the filesystem. For example, it will call write_service_status when a service status change has occurred and it needs to update the file that corresponds to the service in the filesystem.

Next is our init function line:

```
int nebmodule_init(int flags, char *args, nebmodule *handle){
```

It takes three arguments. The first argument is meant to give you some context about how the module is currently being initialized. I don't use it in nagfs. The second argument is a string pointer called args. You may pass arguments to the module using ones found after the module name in the broker_module directive in your nagios.cfg. If you do so, they will be available in this args string. The third argument is a handle to the struct that defines our module. We can use this to refer to ourselves, if, for example, we call a function that requires a pointer back to us. Actually, this happens right off the bat when we register with the broker to get some data:

```
neb_register_callback(NEBCALLBACK_SERVICE_STATUS_DATA,nagfs
_module_handle,0,nagfs_handle_data);
neb_register_callback(NEBCALLBACK_HOST_STATUS_DATA,nagfs
_module_handle,0,nagfs_handle_data);
```

Ask the broker for events with the neb_register_callback function, which takes four arguments. The first is a constant that defines what type of events we're interested in. These are the same constants as in Listing 1. The second is our handle, so that the broker can find out what it needs to find out about us. The third is a priority number. In general, when more than one module registers for the same type of event, they are executed in the order they are loaded by the broker on startup. You can override this behavior by specifying a priority number. The last argument is a function pointer to our data handler function. Our data handler will be the function that actually gets the event struct and does stuff with it.

There's not much more interesting here, so let's skip down to the declaration line for the handler function:

```
int nagfs_handle_data(int event_type, void *data){
```

The data handler must return an exit code in the form of an int and accept two arguments. The first of these is a constant specifying the event type: yes, once again, one of the constants specified in Listing 1. The second is a void pointer, which I'll get to in a moment.

So why would our event handler need to be passed the event type? The event handler function should be able to infer the event type, by virtue of the fact that we specified it when we defined the handler. But notice that we actually use the same handler function for both event types we are registering for. Thus, when the broker spawns the data handler, it passes the event type along, just in case the handler has more than one job (as ours does).

The void pointer is a data struct that is passed from the event broker. It's our magic smoke—the instantiation of the data we're actually looking for. It will be a different type of struct depending on the type of event data the broker passes us. It's up to you to typedef the struct into the correct type. You can find the various types in the broker.c file in the Nagios tarball. Our event handler uses a switch-case on the value of the constant to decide what kind of event we're dealing with. Then it typedefs the data struct accordingly, as you can see here:

```
switch(event_type){

   case NEBCALLBACK_SERVICE_STATUS_DATA:
       ssdata=(nebstruct_service_status_data *)data;
```

In this case, we've gotten service status data, so we've typedef'd the struct into type `service_status_data`. Now I can dereference information about the service from the struct. The broker.c file is also handy for finding out what sorts of data we can dereference from the structs we get from the broker: stuff like `svc->host_name`, which I pass to the write functions I found out about by reading the structs in broker.c.

The rest of the program is pretty self-explanatory. If we get service data, we pass it to the `nagfs_write_service_status` function. Host state data goes to the `nagfs_write_host_status` function. These functions primarily deal with directory and file access and error detection (the "boring stuff").

There are a slew of changes in 3.0 that make the Event Broker even more powerful. My personal favorite is the addition of custom external commands. Basically, these are commands that you make up and pass in to the external command file. They are not processed by Nagios (obviously, since Nagios won't know what you're talking about), but they can be detected and parsed by an event broker module that knows about them, so they're a great way to get external (non-Nagios) data to your module.

The moral of the story is that you should totally write an event broker module. They're fun to write (more fun than writing event handlers in Perl anyway, heh), they'll help other people out (real people, who haven't made exactly the same architectural assumptions you have), and they're a great excuse to dig around in the Nagios source, which I promise you, is some of the most elegant, well-engineered C that you'll come across in a project of this size.

Take it easy.