CERN - European Organization for Nuclear Research
PH/SFT Group - CernVM

iAgent Kernel Documentation

First draft

Author: Ioannis Charalampidis,
Geneva - February 2012

Contents

Kernel Documentation

Introduction

iAgent has an extremely modular design. It's kernel is only responsible for dispatching messages to the registered modules and monitoring their health. All the logic is implemented by the modules. It is extending the asynchronous functionality provided by POE, by introducing a predefined message dispatching technique through the *Module Stack*. This documentation will explain how the kernel works and help you create your own modules.

In order to use this system it is also recommended to read the POE::Kernel documentation from CPAN. (http://search.cpan.org/~rcaputo/POE-1.350/lib/POE/Kernel.pm). Keep in mind that this documentation explains only how the kernel works. The actual logic is implemented by each other module. Please refer to their independent documentation for more details.

Modules

Each module runs in a seaperate POE::Session. The registration of the POE message callback functions to the class instance, and the registration of the session to the Kernel is performed automatically. Additional configuration for the module can be provided through the module's manifest.

An example module follows. This module does nothing but reply to every message it receives (Assuming the XMPP module is loaded). The bare minimal requirements for the plugin is a 'new' sub that will instantiate the class and an optional manifest definition that will specify extra information about the plugin:

```
package Modules::MyModule;
use strict;
use warnings;
use iAgent::Kernel;

our $MANIFEST = {
	config => "myconfig.conf"
};

################################################
sub new {
################################################
	my ($class, $config) = @_;
	my $self = { };
	$self->{config} = $config;

	# Initialize class

	return bless $self, $class;
}

################################################
sub __comm_action {
################################################
	my ($self, $packet, $kernel, $session) = @_[ OBJECT, ARG0, KERNEL, SESSION ];
	iAgent::Kernel::Reply("comm_reply", { data => "You sent: " . $packet->{data} });
}
```

In order to register your module you just need to add the *LoadModule "Modules::MyModule"* directive to *iagent.conf*. The loading process is performed automatically by the kernel and is the following:

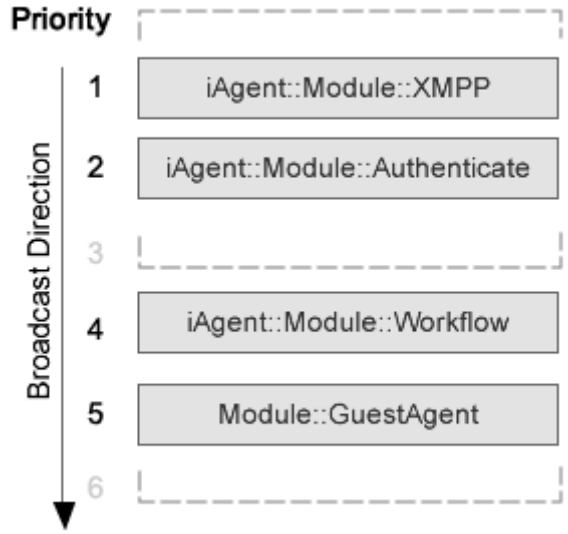
1. If there is a 'config' parameter in the MANIFEST, the kernel will load the specified config file from the *etc* folder and store it in a hash.
2. The kernel will then instance the specified module class passing the config hash as the first argument to the *new* function.

3. If not otherwise instructed by the manifest, the kernel will detect all the functions starting with a double underscore and register them as message handlers for POE messages with the same name. (For example __comm_action will be a handler for the message ‘comm_action’)
4. If not otherwise instructed by the manifest, the kernel will then register the module on the module stack with priority 5.
5. When all the modules are loaded, the kernell will broadcast the message ‘_ready’ to all the registered modules.

Module Stack

When a plugin is registered in the iAgent Kernel it’s session is placed in the Module Stack. The module stack is an ordered stack that is used when a message needs to be broadcasted. The message is sent by traversing the module stack top-to-bottom and sending the message to each one of the registered ‘POE::Session’s. The order of the module on stack is determined by it’s priority. By default each plugin has priority=5 but it can be changed with a manifest parameter.

Each priority is practically a group. Meaning that more than one plugin can have the same priority, but then their order is not explicitly defined. iAgent Kernel ensures the correct order of the groups, but cannot ensure the order inside each priority group.



Usually, the messaging modules go on the highest priorities, followed by the authentication. This way, when a message arrives, it can be filtered or rejected, before reaching the processing modules.

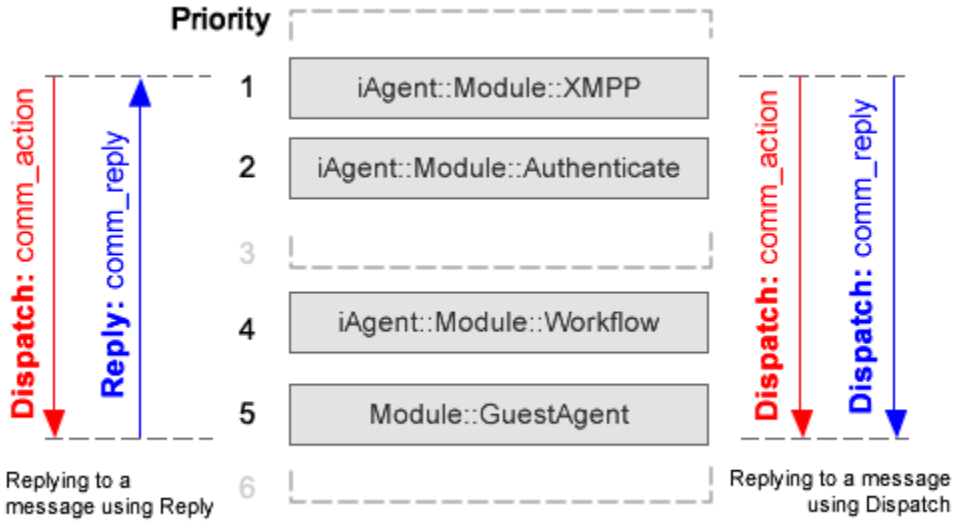
Message dispatching

After all the modules are loaded and the *_ready* message is sent, the kernel is ready to dispatch messages to the module stack. The messages are always sent top-to-bottpm, starting with the modules in lowest priority (Usually 0) to the modules with highest priority. So, the higher the priority, the sooner the module will receive the broadcasted message. Exception in this rule is the *Reply* message. Here are the message broadcasting methods iAgent supports:

- **iAgent::Kernel::Dispatch**(message, [args ..])
  This function will synchronously send the specified message to all the sessions and wait for them to complete before returning. This function is equivalent with the POE::Kernel->call( ) for each session.
- **iAgent::Kernel::Broadcast**(message, [args ..])
  This function will asynchronously send the specified message to all the sessions and return immediately. This function is equivalent with the POE::Kernel->post( ) for each session.
- **iAgent::Kernel::Reply**(message, [args ..])
  This function is similar to dispatch, but is used to reply on messages previously received. This is very useful function because it broadcasts the message bottom-to-up on the module stack, allowing other modules to apply proper transformations. (Like appending a security signature using the authentication module).

Dispatch vs Reply

It is important that, when needed, the Reply function is used instead of another Dispatch, because with Reply you can profit from the reverse traversal of the stack. Take the following example:



When a message arrives from XMPP, the *XMPP* module dispatches a **comm_action** message. If someone wants to reply to this message, he should send a **comm_reply** message to the XMPP module. Let's say now that we need trusted communication. For this purpose we install the Authenticate module that signs each outgoing message and verifies the signature of each incoming message. To do so, the Authenticate module listens for **comm_action** messages, checks the 'signature' attribute, and verifies the data with it. Also, it listens for **comm_reply** and **comm_send** messages, calculates the signature and injects it in the message.
If we use again *Dispatch* instead of *Reply*, the **comm_reply** message will not reach the Authenticate module in correct order and the signature will be missing from the final packet.

- **iAgent::Kernel::Query**(message, [args ..])
  This is a function is simmilar to Dispatch, but it collects all the responses from each module and returns an array with all of them.

Another important feature of the message dispatching is the fact that every module can halt the execution

Error handling

In order to be secure and efficient, the Kernel needs to monitor all the modules for their health and perform the appropriate recovery sequence if something goes wrong. Since the kernel is using POE, a session will automatically exit when there are no pending messages. The kernel can detect such cases and perform the action specified by the manifest. There are three cases:

- **oncrash => “reload”** : In this mode the kernel will try to reload the crashed module. If it fails, it will terminate the application.
- **oncrash => “die”**: In this mode, if the module crashes, the entire application must be killed. This is usually used in authentication and security modules.
- **oncrash => “restart”**: In this mode, if the module crashes, the application must be restarted.
- **oncrash => “ignore”** (Default): In this mode, a module crash will not trigger anything.

There are however some cases, a module needs to trigger a fatal error (Like ‘unable to connect’ on XMPP). This should not be done by letting the session to exit, but using the **iAgent::Kernel::Crash**(message) function. This function will trigger the appropriate crash sequence, but also perform a crash dump that might help troubleshoot a problem.

Appendix

A. Kernel messages

Besides the very basic messages sent by POE::Kernel (_start, _stop, _child, _parent), the following POE messages are also sent to sessions by the kernel:

- **ready( )**
  Sent by the Kernel when all the modules are loaded and all the sessions are started.
- **_setup**()
  Sent by the kernel in order to perform initializations of the module that require a properly set-up POE Kernel.
- **_stop**( )
  Sent by POE after the kernel forces the session to shut down (by signaling SIGINT on every session). You can safely use this message for cleaning up your module.
- **_recover** (HASH)
  Sent by the Kernel when the module was successfully recovered after a crash. The first argument is the second argument passed to *iAgent::Kernel::Crash*. It is intended for internal use by the plugin.

B. Manifest parameters

Here is a reference to the manifest parameters that kernel accepts

- **config** [string] (default: “”)
  Specifies the configuration file (relative to iAgent’s etc folder) that should be loaded for this module.
- **priority** [int] (default: 5)
  Specifies the priority of the module in the module stack. The higher the priority, the sooner a broadcast message will reach the module.
- **oncrash** [string] (default: “ignore”)
  What action should be performed if this module crashes. Possible options: *reload, restart, die, ignore*
- **hooks** [string | array | hash] (default: “AUTO”)
  This variable defines the message handlers for every POE message that arrives on the module’s session. It can be one of the following:
  - **‘AUTO’** (string)
    In this mode, the kernel will scan the functions of the module and will register the ones that their name start with the ‘hooks_prefix’ (see below) as handlers for the same message. (For example: *__comm_action* will be a handler for message *comm_action* if hooks_prefix is ‘__’)
  - **[‘name’, ‘name’, ‘name’, ...]** (Array)
    In this mode, the kernel will assume that for every message specified, there is a function in the class with the same name that handles it.
  - **{ “message” => “handler”, “message” => “handler” … }** (Hash)
    In this mode, the kernel will register the specified handlers to the specified POE Messages
- **hooks_prefix** [string] (default: “__”)
  This variable defines the prefix of the functions that will be automatically chosen as message handlers, when hooks is set to ‘AUTO’

Keep in mind that other modules might handle heir own directives in other module’s manifests. For example XMPP module provides automatic handling of XMPP messages, defined through the *XMPP* manifest parameter.