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

iAgent Introduction Guide

First draft

Author: Ioannis Charalampidis,
Geneva - March 2012

How do I...

Hello! This is an introduction guide to the iAgent framework. In this document we will answer step-by-step all the questions required in order to have a complete idea on how to extend this system. Here are the questions answered:

1. How do I install iAgent?

Step 1 - Prerequisites

First of all, you need to have perl, so depending on your operating system, install Perl 5.8.0 or later to your machine.

Step 2 - Perl Modules

Then you need to ensure the following CPAN perl modules are installed:

- POE
- Config::General
- Term::ANSIColor
- Net::XMPP
- JSON
- XML::Simple
- DBI
- DBD::SQLite
- Hash::Merge
- Data::UUID
- Net::LDAP
- Getopt::Long
- POE::Component::Client::HTTP
- POE::Component::SSLify

If you have Perl properly installed you would be able to do so by invoking the following command:

```
sudo cpan POE Config::General Term::ANSIColor Net::XMPP\
          JSON DBI DBD::SQLite Hash::Merge Getopt::Long
          POE::Component::Client::HTTP POE::Component::SSLify DateTime
```

If you don't have CPAN and you have a special reason not to install it, you can try your luck using yum:

```
sudo yum install perl-POE perl-Net-XMPP perl-JSON perl-DBI perl-DBD-SQLite perl-Hash-
Merge perl-Getopt-Long-Descriptive perl-DateTime perl-Config-General perl-XML-Simple
perl-Data-UUID perl-LDAP perl-File-ReadBackwards perl-POE-Component-SSLify perl-
Component-Client-HTTP
```

Note: Since this document is not always up-to-date, if you get a message from perl like "Can't locate Some/Module.pm in @INC", try to look-up that module on CPAN or on yum (`sudo cpan Some::Module` or `sudo yum install perl-Some-Module`).

Step 2 - Download

iAgent has no concrete installation. You just need to fetch the latest sources which commonly contain the following folders:

- `bin : Contains the bootstrap scripts`
- `lib : Contains the sources of iAgent`
- `etc : Contains the configuration`

Step 3 - Configure Core

Then you need to decide what you need to load and what remaining configuration is required. To do so, open the etc/iagent.conf and look at the LoadModule directives. A fairly good starting point is the following:

```
LoadModule "iAgent::Module::XMPP"
LoadModule "iAgent::Module::LDAPAuth"
LoadModule "iAgent::Module::CLI"
```

This will load a communication module (XMPP), an authentication module for the communication module (LDAPAuth) and a command-line interface (CLI).

Step 4 - Configure Modules

You then need to tune the appropriate configuration files for the loaded modules. In our case you need to edit the following files. For more details refer to the documentation of each module:

Edit the **xmpp.conf:**

```
# XMPP Server information
# Edit the following to match your configuration:
XMPPServer       "my.server"
XMPPUser         "myuser"
XMPPPassword     "s3cret"
XMPPRegister     1

# VCard information
<XMPPVCard>
    # That's important!
    role         "builder"
</XMPPVCard>
```

Then edit the **ldap.conf:**

```
# LDAP Authentication
# Edit the following to match your configuration:
LDAPServer       "mydc.company.com"
LDAPBindDN       "cn=admin,OU=Users,OU=Organic Units,DC=compay,DC=com"
LDAPBindPassword "my-secret-password"
LDAPSearchBase   "OU=Users,OU=Organic Units,DC=company,DC=com"
LDAPNameAttrib   "cn"

# Map LDAP groups to permissions
<Permission permission_getown>
    LDAPGroup    "CN=reading-users,OU=e-groups,OU=Workgroups,DC=company,DC=com"
    Default      1
    Description  "User can check his permissions"
</Permission>

# ...
```

2. How do I start iAgent?

Step 1 - Run

Starting iAgent is simple. You just need to enter the bin directory and invoke the ./start.sh script. If you need more debug information, change the Verbosity parameter of iagent.conf to a lower value.

3. How do I write my own iAgent module?

Step 1 - Prerequisites

First thing, you need to know how POE works, since iAgent wraps around it. So, have a look here: http://search.cpan.org/~rcaputo/POE-1.350/lib/POE.pm.

Step 2 - Boilerplate

You can start by copying the following boilerplate to the file lib/Modules/**MyFirstModule**.pm :

```
package Module::MyFirstModule;

use strict;
use warnings;
use POE;
use iAgent::Log;
use iAgent::Kernel;

############################################
# Manifest definition
our $MANIFEST = {
############################################
};

############################################
# Create a new instance
sub new {
############################################
    my $class = shift;
    my $self = {

    };
    return bless $self, $class;
}
```

Step 3 - Loading your module

Then you need to edit etc/iagent.conf and add the LoadModule “Module::MyFirstMmodule” directive.

If you invoke bin/start.sh now; if you see the message “Plugin Module::MyFirstModule loaded” congratulations! You have now your first module successfully loaded on iAgent.

Step 4 - Adding message handlers

Let’s start involving some logic now. Previously, I said you need to know how POE works. That’s because every module runs on it’s own POE::Session and they communicate with each other through the iAgent Kernel. The easiest way to register your message handlers for your POE Session is the following:

```
sub __my_cool_message {
}
```

Yup, just by defining a sub prefixed with “__” you automatically define a hook handler for the message “my_cool_message”. If you prefer the clasic POE-Way of defining handlers, this feature is fully customizable through the module’s manifest. Refer to the kernel documentation for details.

Let’s handle now a message that comes from the iAgent when everything is loaded. The syntax of the function is exactly the same as the official handling functions of POE messages, so you have $_[KERNEL], $_[SESSION] and the rest of the POE arguments passed in your function:

```
sub __ready {
   $_[KERNEL]->delay( late_message => 1 );
}
```

Step 5 - Speak with other modules

So far you saw how you can handle incoming messages. The messages arrive there either through a message sent by your module targeting itself, or through the iAgent Kernel. Every time you send a message to the kernel it passes through every module in a predefined order. There is no way to directly speak with another session and that’s something that was intentionally avoided. You will see later why...

In order to dispatch a message to all of the plugins you can use the iAgent::Kernel::Dispatch function:

```
sub __late_message {
   iAgent::Kernel::Dispatch(“cool_command”, { temperature => -40 });
   return RET_OK;
}
```

The message cool_command will be dispatched to all of the modules in an order defined by their priority in the module stack. Keep in mind that it also includes your module, so try to avoid infinite loops!

It’s also important to always return a proper value from your message handler. Returning RET_OK will allow the further propagation of the message, while RET_ABORT will block it. (Refer to the kernel documentation for details).

So, let’s handle your message in the same module:

```
sub __cool_command {
   my $hash_ref = $_[ARG0];
   iAgent::Kernel::Dispatch(“cli_write”, “It’s $hash_ref->{temperature} degrees
cool!”);
}
```

If you run iAgent now you will see the message “It’s -40 degrees cool!” after a 1 second delay.

Step 6 - Intercept messages

An important functionality of iAgent is the ability to intercept and modify dispatched messages on the fly. Doing so, allows you to extend the functionality of a module without messing with it’s source. For this test case we will create another module that will intercept the cool_command message and change the temperature:

```
package Module::MyFirstIntercepter;

use strict;
use warnings;
use POE;
use iAgent::Log;
use iAgent::Kernel;

############################################
# Manifest definition
our $MANIFEST = {
############################################
    priority => 4
};

############################################
# Create a new instance
sub new {
```

```
############################################
    my $class = shift;
    my $self = {

    };
    return bless $self, $class;
}

############################################
# Intercept cool_message
sub __cool_message {
############################################
    my $hash = $_[ARG0];

    # Warm it up a little
    $hash->{temperature} += 10;

    # Return RET_OK to allow further propagation of the event
    return RET_OK;
}
```

Two things you should note here:

1. The **'priority'** parameter in the manifest hash tells iAgent to load this module under priority group 4. By default every module is loaded to priority group 5. The higher the priority, the sooner the message will arrive to that module.
2. The **RET_OK** of the message handler, that allows the further execution of the message. If you remove the return statement, or if you change the value to RET_ABORT, the message "cool_message" will never reach the previous module.

Edit your iagent.conf and instruct iagent to load the "Module::MyFirstIntercepter" module and start the application again. You will now see the message "It's -30 degrees cool!" after a 1 second delay.

4. Where can I find what messages can I intercept/handle?

There are two ways: You can either see the documentation of each module, or browse the source of each module and detect message broadcasts by one of the following statements:

- iAgent::Kernel::Dispatch(MESSAGE, …)
- iAgent::Kernel::Broadcast(MESSAGE, …)
- iAgent::Kernel::Query(MESSAGE, …)

There are also some standard messages broadcasted by the kernel and they are the following:

- _ready : Sent when the iAgent has finished loading modules and is ready to run
- _start : Called by POE as the entry point of your application
- _stop : Called by POE to clean-up before termination

5. Where can I find what messages can I dispatch?

There are two ways: You can either see the documentation of each module (recommended), or browse the source of each module and detect message handlers. That are usually functions that start with '__' prefix, but there are some more complex cases.

6. Where is the _start message handler?

It is true that _start is the first event sent by POE to every session. If the session doesn't do anything inside this event, the session will automatically die right after. Since the modules in iAgent kernel are highly asynchronous and they usually don't have any 'event loop', they need an infinite loop to keep them alive.

This loop is secretly created by iAgent kernel after instantiating your module as an in-line state handler:

```
_start => sub {
     $_[KERNEL]->delay( ___dummy_loop___=> $LOOP_DELAY );
},
___dummy_loop___ => sub {
     return if (defined ($_[HEAP]->{_KILLED}));
     $_[KERNEL]->delay( ___dummy_loop___=> $LOOP_DELAY );
}
```

That's why you don't see any _start handler. However, if you NEED to implement a start handler you are free to do so, but then you are the one responsible of keeping your session alive.

7. How do I define XMPP message handlers?

In order to be easier to use, XMPP and other modules provide additional configuration through each module's idividual manifest. For example, in order to avoid writing the same filtering, permission checking and parameter validating routines for every module, XMPP handles the following configuration from the manifest:

```
############################################
# Manifest definition
our $MANIFEST = {
############################################

   XMPP => {

      "<message context>" => {
         "<message type>" => {
            "<action>" => {
               message => "my_action_handler",
               parameters => [ 'from', 'to' ],
               permissions => [ 'read' ]
            }
         }
      }

   }

};
sub __my_action_handler {
    my $packet = $_[ARG0];
    my $from = $packet->{from};
    my $parameters = $packet->{parameters};

    # ...

    return RET_OK;
}
```

See the XMPP module documentation for more detailed description of the manifest-based configuration.

8. How do I add my own CLI commands?

Like the XMPP module, CLI module provides the same configuration technique through the module's manifest.

```
############################################
# Manifest definition
our $MANIFEST = {
############################################

   CLI => {
      "<group>/<command>" => {
          message => "my_command",
          description => "A few words",
          parameters => [ 'name=s', 'f' ] # (Getopt-compatible syntax)
      }
   }

};

sub __my_command {
    my $cmd = $_[ARG0];
    my $parameters = $cmd->{parameters};

    return RET_COMPLETED;
}
```

<u>Something important</u>: When a CLI command is invoked, the CLI does not return to the prompt until it receives the cli_completed message OR if the command handler returns with a RET_COMPLETED. So, if you have an action that takes time to complete and it's asynchronous, you must return with a RET_OK after the invocation and dispatch cli_completed when you are done.

Example:

```
sub __my_command_handler {
    my $cmd = $_[ARG0];
    iAgent::Kernel::Dispatch("do_stuff", { parameters => $cmd->{parameters} );
    return RET_OK;
}

sub __stuff_completed {
    iAgent::Kernel::Dispatch("cli_completed", 0); # ARG0 is the return code
}
```

9. How do I access other module's manifest?

If you are writing your own manifest-processing module you might need to access every module's manifest. You can find all the currently running modules in their appropriate order in the module stack in the @iAgent::Kernel::SESSIONS array.

Each element of the array is a hash in the following structure:

```
{
    class => STRING,             # The name of the package of the module
    instance => OBJECTREF,       # A reference to the instance of the package
    priority => NUMBER,          # The priority of the module
    session => OBJECTREF,        # The POE::Session of the module
    manifest => HASHREF          # The hash reference to the module's manifest
}
```

So, in order to collect all the extra configuration of each manifest you heed to wait for the _ready event and then process the manifest of all the running sessions.

10. How do I exit iAgent?

Since there are many modules running and their state is not always known, it's not good idea to exit from iAgent using die or exit. The best way to exit iAgent is though the following command:

```
iAgent::Kernel::Exit( <exit code> );
```

This command will gracefully ask each module to shut down and eventuall quit.

11. How do I write my own workflow handler?

If you want to create your own handler of a workflow action you just need to define it in the manifest of your module:

```
############################################
# Manifest definition
our $MANIFEST = {
############################################

   WORKFLOW => {
      actions => {

          “<my awsome action>” => {
              run => “<external program or function name>”,
              mode => “exec” or “thread”,
              description => “<a human-readable description>”,

              # Optional stuff
              handle_validation => “<POE message that will validate the context>”,
              handle_input => “<POE message that prepares the input for the action>”,
              handle_output => “<POE message that post-processes the output>”,
              handle_logs => “<POE message to collect the logfiles>”
          }

      }
   }

};
```