# Comp 2207 - Coursework

SMART METERING SYSTEM

CONOR KEEGAN - 26100428

# Introduction

## Starting the Smart Metering System

The Smart Metering System is split up into four different programs that do different things, the first and most important being the Server. The Server program should be run before all of the other programs, as all of the others will wait for it to be started.

The first step is to start the Server class file, it has the main method that will deal with all of the other Server classes. You will not need to start rmiregistry externally as the program will attempt to do this for you. The server GUI will pop up and the console will output 'Server ready' when it has started.

Next you can start any of the other programs and multiple if you wish. For the Power Company and Broker programs, you can add a program argument of the names that you would like to name them. This is not necessary however as if no argument is specified then it will just be given a number increasing from 1, as there are more programs opened. The Meter program is not like this as it will automatically get a number to refer to it, so there is no point specifying a program argument.

## Using the Smart Metering System

Each program has its own GUI so it can update the user with useful information, and for some of the programs to allow for use input. The Server GUI only contains a console which gives updates on registrations and lookups. The Meter GUI also has the console, but also allows for user input. From here you can register to random Power Companies, unregister from them, request deals and other things to show that the classes are communicating. The Power Company has the console but only allows for a changing of the tariff price, and the Broker is exactly the same as the Server. To close the programs, all you have to do it exit the GUI, this will terminate the program.

# Smart Metering solution classes

The Classes for the Smart Metering solution are split into four different classes, the Server's classes, the Meter's classes, the Broker's classes and the Power Company's classes. Each of these sets have a main class named Server, Meter, PowerCompany and Broker respectively, which contain the method that allows them to be run. All of the classes talk to each other through the use of Java RMI and so they all have objects that have remote methods that can be called by the other classes.

## Design and Implementation

### Server

The Server classes are used to allow the other classes to locate each other in an easy manner and have it confined to a single location. This means that each group only needs to know the static reference of the server, as it can then get the references to the other objects from it.

### *Server*

This class is the main class and the class that contains the main method which starts and runs the server. When it is started it first makes sure that there is a running rmiregistry and starts one if there is not. The rmiregistry is the application which allows the server to have a remote version of its object running, and allows other classes to use it. It then creates and instantiates a ServerObject, and binds that to the rmiregistry. This allows the object to be accessed if you have the URI for the object (in this example "rmi://localhost/Server"). The URI to the Server is hard coded into the code, as it will always need to have a link to the Server. If it was going to be implemented in reality, the 'localhost' would most likely be replaced by a URL, so that it would not be limited to one IP address, and you could change the IP address and would not need to change the code. It then starts its GUI so that you can see what the object is doing.

### *ServerObject*

This class is used to keep track of all of the Meters, the Power Companies and the Brokers and their respective location. There can be only one instance of this class on the rmiregistry. This class maintains a list of all of the meter's serial numbers against their location, all of the power company's names against their location and all of the broker's names against their location. It uses the interface RemoteServerInterface to allow the classes to call methods on it.

- getMeter(), getPowerCompany() and getBroker()
  These methods perform a lookup into the objects Hash Maps to get the Remote Objects name from the name of the different program. They then will perform a lookup in rmiregistry to get the reference to the object and return it. They have been designed so that they are able to look up all of the objects and deal with any connection problems, by retrying until they work. Meaning that all of the lookups are kept to a single program.

- getBrokers() and getPowerCompanies()
  These methods are used to get a list of all of the currently existing objects, which is useful in the broker where it needs to find the best deal for a meter out of all of the possible Power Companies. It does this by getting the key set of the Hash Map that contains all of the objects.

### *RemoteServerInterface*

This interface is used in order for the objects to be able to call the ServerObject methods remotely using Java RMI.

### *ServerGUI*

This class is used in order for the user to be able to keep track of what is going on in the Server, as it displays a log of all the important events that have happened on the server along with a timestamp.

## PowerCompany

The PowerCompany class is the class that allows the Meter to be able to use power. It has Meters that are registered to it and send it readings on a time basis. This would allow a Power Company to always keep up to date with the usage of each person that is registered with their company. It is also able to receive alerts when a Meter has been tampered with or an alarm is raised in the Meter. This would allow the company to know that they need to send someone out to either check and fix the Meter, or to talk to the occupant of the house and question them over tampering with the Meter.

### *PowerCompany*

The PowerCompany class is designed as a standalone program that connects to the Server to allow it to get the references to the Meters and the Meters to do the same to it. It firsts makes sure that there is an instance of rmiregistry running, which would be needed if it was running on multiple machines. This is because the rmiregistry is needed to be able to access the object from another computer. It will then attempt to access the Server object, if it fails it will keep retrying after a delay until it either connects or it is terminated. Once it has a copy of the Server object, it checks the program arguments and if it finds one it uses it as the name of the power Company, or gets a number to use from the Server if there are no arguments. It will then register its PowerCompanyObject with the Server by telling it the reference to it. Finally it will start the GUI so that the log can be seen and the Tariff price can be changed.

### *PowerCompanyObject*

This class is used to enable the recording of Meter readings that it is sent, to register a Meter and to theoretically supply it with the utility. It contains a list of all of the Meters that are currently registered to it, as well as a list of all of the Meters that were ever registered to it and their corresponding readings history. It is also able to supply its tariff information to a Broker so that it can find the best deal for a Meter. There can be multiple Power Companies running on the same computer, or multiple computers at the same time. However, all of the names for the Power Companies must remain unique to only that one.

- sendCommand()
  This method is used to send a command to a currently registered Meter. This could be used for lots of different things, such as to ask the meter to send a reading to it, or it could be to disconnect the power from the client due to a lack of bill paying. This is implemented by sending a String command to the Meter. The Meter would then execute this command. The execution of commands is not currently implemented in this solution.
- registerMeter() and unregisterMeter()
  These methods are meant to do exactly what they say, they register and unregister a Meter from the PowerCompany. This would be used to switch Power Company, or to initially register with a Power Company. It is implemented by either adding or removing the Meter from the list that is maintained by the PowerCompany of the currently registered Meters.
- receiveReadings()

This method is used to allow the Meter to send its readings to the PowerCompany, since RMI is used there needs to be a method where the readings are passed as an argument that the Meter can call. This method then saves the readings that are received into a Hash Map that links it to the Meter that it was sent from.

- receiveAlert()
  This method allows the PowerCompany to receive an alert from the Meter in the same way that the readings are received. Currently nothing is done with the alert that is received, however in a practical environment it would then flag it somewhere, where it can be then processed by someone.
- getTariffInfo()
  This method enables the PowerCompany to send the Broker its tariff info so that it can be checked to see if it can provide a better deal to a possible client.

### *RemotePowerCompanyInterface*

This interface is used to enable methods to be remote when using RMI, all of the PowerCompanyObjects remote methods are able to be called remotely by the other objects.

### *PowerCompanyGUI*

This class is used in order to display a log of everything that is happening with the object. It will display useful information about the running of the PowerCompany. Its displays information such as registering, unregistering, and receiving readings and alarms. It also puts a timestamp on each log entry. This class also provides a way for the user to change the Tariff price of the PowerCompany so that it is more competitive.

## Broker

This program is used to enable a Meter to check if there is a Power Company that is offering a better deal than is currently being given to them. It does this by receiving a request from a Meter to get a new deal, and then contacting all of the Power Companies and deciding based on information from the Meter what is the best deal. It will then send that deal back to the Meter. The Meter will then send back a response, as to whether to accept or reject the deal. If it accepts, the Broker will then facilitate the transfer of Power Companies for the Meter.

### *Broker*

The Broker class is used as the standalone part of the Broker as it contains the main method that runs the Broker. It works in almost the same way as the Power Company, as it gets a makes sure that rmiregistry is running, and then makes sure that it gets a copy of the Server object. Once it has these, it checks to see whether it has received any program arguments, if it has uses this as the name for the Broker, if not it gets a unique number from the Server to use as its name. It then creates the BrokerObject and binds it to the rmiregistry and sends the name of the object to the server, and finally starts the GUI.

### *BrokerObject*

This class performs all of the actions that are needed in finding a new deal for a Meter and then changing the Power Company for the Meter. It maintains a Hash Map of all of the deals that it has sent out that are waiting responses from, so that when a Meter sends its response, it has the deal needed to implement the appropriate actions. There can be multiple Brokers in the same way as there can be multiple Power Companies, as in they all have to have different names.

- findDeal()
  This method is the main part of this class as it performs the majority of the actions. The first thing it does is get a reference to the Meter that requested the deal. It then uses this reference to get the readings history from the Meter to find the most appropriate tariff. It then gets references to all of the Power Companies, and individually checks them and then selects the best from them. From this data it then constructs the deal with all of the necessary information to switch the Meter if the deal is accepted. The last thing that it does is send the deal back to the Meter and records puts the deal into a Hash Map referencing the Meter that it relates to.
- acceptDeal() and rejectDeal()
  Both of these methods first check that the Meter that is trying to accept a deal has a deal related to it in the Hash Map. The accept method then gets the reference to the Meter and unregisters it with the old Power Company and registers it to the new Power Company. The reject deal just removes the Meters entry in the Hash Map, so there is no issues if a new deal is requested.

### *RemoteBrokerInterface*

This class is used to enable the Meter to be able to call the BrokerObject methods remotely, using RMI.

### *BrokerGUI*

This class works exactly in the same way as the ServerGUI as it only displays a console which outputs useful information with a timestamp about what the Broker is doing.

## Meter

This is the main program for this system as it does all of the monitoring of the utility usage and the sending to the Meter. It has to actively update the reading so that it can maintain the correct reading, as well as periodically updating the Power Company with the usage that it has been recording. It has to be able to contact the Power Company with those readings, it also has to be able to report any alarms that may arise. It is also able to receive and run commands from the PowerCompany remotely.

### *Meter*

The Meter class is the class that runs the Meter by implementing all of the other classes into the standalone program. It, like the others, checks for an rmiregistry and starts one if necessary. It then tries to get the ServerObject until it can get it. Once it has the server object, it uses it to get a serial number to use as the Meters name. It uses this to bind a MeterObject to its rmiregistry, and sends the reference to the Server. It then starts the GUI

and the threads that update the reading and periodically send the readings to the PowerCompany, if it is registered to one.

### *MeterObject*

This class contains most of the implementation of the Meter and all of its methods, and remote methods. It is able to send readings and alerts to a registered PowerCompany, request deals from a broker as well as run commands sent from a PowerCompany. There can be multiple Meters connected to the Server and to the power Companies as well as the Brokers.

- sendReadings() and sendAlert()
  These methods first checks whether the Meter is registered to a Power Company, as it can only send readings and alerts if this is the case. It then gets a remote reference to the Power Company, and sends its readings/alert to it, and will retry after a timeout period if it cannot connect to the PowerCompany.
- requestDeal()
  This method will try to connect to the Broker that is supplied to it, and will keep retrying until it can. When it has connected, it will then call the remote method on the Broker and will then return as it has nothing to do with the receiving of the deal.
- acceptDeal() and rejectDeal()
  These methods will get a reference to the current Broker that is being used to find a deal, and will return if there has been no requested deal. It will then us the reference that it has to connect to the Broker and call the respective method on the Broker.
- registerPowerCompany() and unregisterPowerCompany()
  These methods are remote methods so they can be called by the Broker when it is getting a better deal for the Meter. It will get a reference to the Power Company that it is registered to if it is registered to something, and will call the unregister method on it and forget the reference to that Power Company. If it is registering it will then get the reference to the new Power Company and call the register method on it, it will then remember the reference to the new Power Company.
- runCommand()
  This remote method will run the command that is supplied as an argument to the method.
- getReadingsHistory()
  This remote method will return the current readings history of the Meter, for use by the Broker in finding a better deal for the Meter.

### *RemoteMeterInterface*

This class is used to enable to Meter to have remote methods that can be called by the other classes in order to be able to get information or affect the Meter.

### *MeterGUI*

This class is used in order to display the information of the Meter and to accept user input to affect the Meter. It displays the currently registered PowerCompany as well as the current reading which updates every second. It also has a console like the other GUIs to

show what the Meter is doing. It has input for the user to register with a random PowerCompany as well unregister from one. It has input to request, accept and reject a deal and a button which simulates tampering with the Meter.

#### *MeterReadingThread*

This class is a Thread which increases the current reading of the Meter with a random number every second.

#### *MeterSendResultsThread*

This class is a Thread which will periodically send the current readings of the Meter to the Power Company if the Meter is registered to a PowerCompany.

## Testing

### Server

To test the Server I started using the other programs to see if they worked correctly, from the console output on the GUI you can see that it is responding correctly (Figure 7).

### Meter

To test the Meter at registering I tested it with registering and unregistering with two different Power Companies (Figure 1). To test the request deal, I do the same but now with being registered to one company, requesting a deal (Figure 2) and accepting to show that it will change the company (Figure 3). To show that the Alert is sent, I press the Tamper button which sends an alert and show that it is received by the PowerCompany and you can also see from the periodic readings sending that that also works (Figure 4 and Figure 5).

### PowerCompany

To test the PowerCompany, all that is needed is to see that it is receiving all of the requests from the Meter (Figure 5), and that it is responding to the Brokers requests for information (Figure 6).

### Broker

To test the Broker all that needs to be tested is if it can receive a request, deal with it, and change the Meters Power Companies (Figure 2, Figure 3 and Figure 6).

## Meter changing Power Company via Broker

The Smart Metering Solution allows the Meter to change Power Companies via a Broker due to power of using RMI. The way that it works is that, when the Meter accepts the deal from the Broker, it calls the accept method on the Broker. The Broker then gets from the Server a reference to the Meter, it is then able to call the remote methods of the Meter. One of those methods being the unregisterPowerCompany(). So the Broker first unregisters the Meter from the Power Company that it is changing from. The Meter then contacts the Power Company and unregisters from it, and then forgets the reference to the Power Company as it doesn't need it any longer. The Broker then calls another method registerPowerCompany() on the Meter, and passes the name of the new Power Company to it. The Meter then contacts the new Power Company and registers itself with it and

remembers the reference to it. This is demonstrated through Figure 2, Figure 3, Figure 4 and Figure 6.

## Multiple Object Connections

A Power Company can be connected to multiple Meters due to the way that RMI works, since all of the Meters can have a stub copy of the Power Company class locally. To the class it is not calling a remote method, it is just calling a local method. It is the underlying RMI middleware that will deal with all of the network connections. RMI also starts its own threads in order to deal with multiple connections to the same method unless they are synchronized. Since all of the Meters do not need to be connected to the Power Company all of the time, the Power Company can deal with all of the Meters. The Broker can be connected to multiple Meters and power Companies in the same way, since they only need to use the references to the classes it can talk to them all.

## Lost Connections

Lost connections are dealt in a few different ways in this solution. The first way is a way that I have implemented into the code. I have made it so that whenever the there is a connection issue where the method being called, or the object being looked up, the code will wait an amount of time and then try again until it works or the program is terminated. Another way that I have tried to handle connection issues is the way that I have implemented the Server. The Server class is the only one that will look up the references to the objects, because it has the full list of all of the objects. This means that if a program crashes and is restarted, it can still connect to that object because it will look up the object. If it just had a list of references to the objects and not the location of them, it would be using an invalid reference to an object that doesn't exist anymore. RMIs connections will handle some lower level connection issues such as packet loss. Since it uses the TCP protocol, and receives acknowledgements of packet arrival, it knows when it is necessary to resend packets. For this to work effectively on my solution, it would require a lot more tweaking of exception catching and retrying. It would also require there to be a persistence layer for the data so that the programs would be able to restart, and then carry on from where they stopped. All of this has not been implemented, but it would be necessary to avoid a lot more of the connection issues and crashes.

## Implementation on multiple Machines

Implementation of the ability to use on multiple machines would not be very difficult to implement in my solution. To do it you would have to alter the reference name that the programs sent to the server class. It would need to be changed from just the name to the IP address of the machine forward slash the name. The Server code would then just need to be change so that it would only need to add 'rmi://' to the code instead of 'rmi://localhost'. On a LAN this would work perfectly in nearly all cases and could work well across the internet. If persistence was added, when restarting the IP address would have to be checked against the Server to make sure that it hadn't changed.

## Conclusion

The use of Distributed Objects in the solution to the Smart Metering System worked well. All of the applications that are needed fit nicely into these objects with the remote methods that come with them. With the Distributed Objects, the communication between the different applications is unknown to the application itself, meaning the networking is not specifically developed for that implementation. This makes this suitable to upgrade a system that previously did not link at all. So with the objects still be separate, but linked through hidden communication seems to be useful. This allows for the upgrade to still keep the key parts of the system, and for all of these parts to be linked together. The Distributed Objects also allow the implementation to move away from the standard Client/Server set up that most modern distributed systems follow. Where everything is routed through a server, whereas is doesn't need to be in this model. The objects can connect to and talk with the specific objects that they need. With other implementations, the addition of other objects, or new resources are a potential cause for problems. But with Distributed Objects, you can easily add and replace things. This allows the system to be more dynamic and to change and adapt as necessary to the environment that it is in.

Distributed Objects do have their own problems, such as the multiple points of failure that they can have. With lots of different machines being part of the network, issues occur when one of these crashes, or is overloaded. Meaning that there has to be a lot more consideration on accounting for crashes, or objects missing or inaccessible. It also adds a huge amount of complexity to the model of the system, with objects linking to other objects, which in turn link to other objects. It could cause problems when implemented on a big scale.

Overall, the use of Distributed Objects in Smart Meter System implementation is better than not using it. With the ability of it to adapt and to scale based on need. Having a system such as this being able to be this dynamic is a very useful thing to have. It does have issues with multiple points of failure, and with having an increased complexity in the system. But these issues can be dampened and can be made easier to manage.

# Appendices

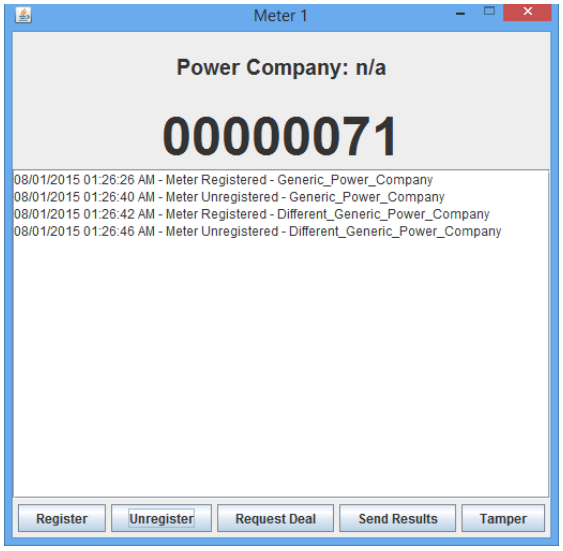


*Figure 1*

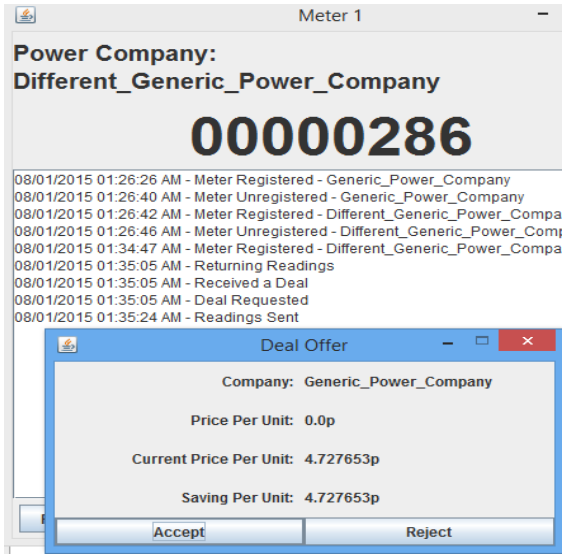


*Figure 2*


*Figure 3*

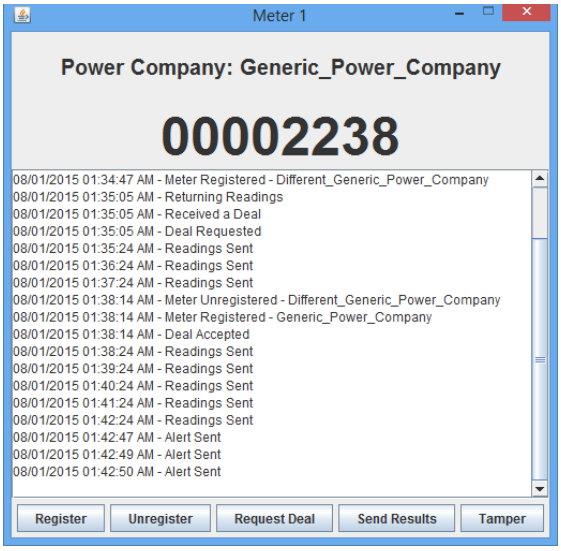


*Figure 4*

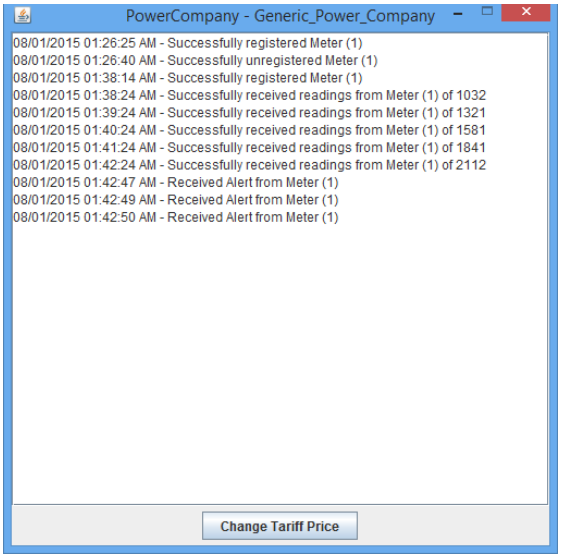


*Figure 5*

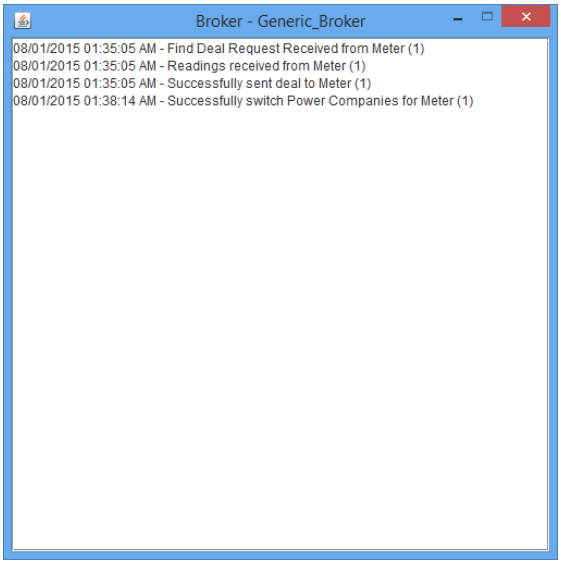


*Figure 6*

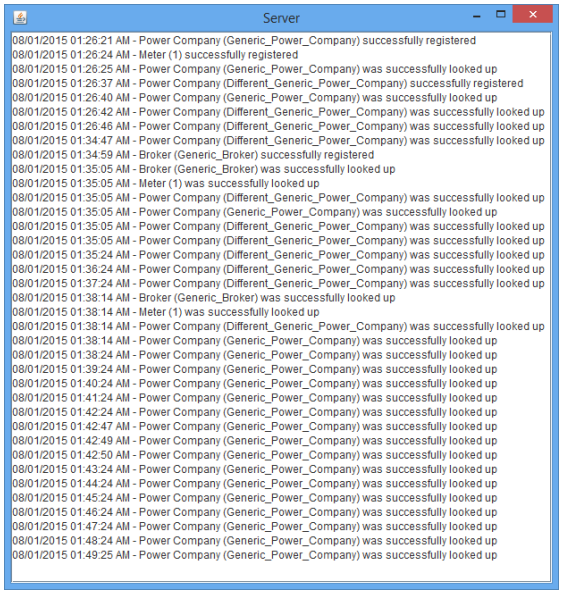
Server
08/01/2015 01:26:21 AM - Power Company (Generic_Power_Company) successfully registered
08/01/2015 01:26:24 AM - Meter (1) successfully registered
08/01/2015 01:26:25 AM - Power Company (Generic_Power_Company) was successfully looked up
08/01/2015 01:26:37 AM - Power Company (Different_Generic_Power_Company) successfully registered
08/01/2015 01:26:40 AM - Power Company (Generic_Power_Company) was successfully looked up
08/01/2015 01:26:42 AM - Power Company (Different_Generic_Power_Company) was successfully looked up
08/01/2015 01:26:46 AM - Power Company (Different_Generic_Power_Company) was successfully looked up
08/01/2015 01:34:47 AM - Power Company (Different_Generic_Power_Company) was successfully looked up
08/01/2015 01:34:59 AM - Broker (Generic_Broker) successfully registered
08/01/2015 01:35:05 AM - Broker (Generic_Broker) was successfully looked up
08/01/2015 01:35:05 AM - Meter (1) was successfully looked up
08/01/2015 01:35:05 AM - Power Company (Different_Generic_Power_Company) was successfully looked up
08/01/2015 01:35:05 AM - Power Company (Generic_Power_Company) was successfully looked up
08/01/2015 01:35:05 AM - Power Company (Different_Generic_Power_Company) was successfully looked up
08/01/2015 01:35:05 AM - Power Company (Different_Generic_Power_Company) was successfully looked up
08/01/2015 01:35:24 AM - Power Company (Different_Generic_Power_Company) was successfully looked up
08/01/2015 01:36:24 AM - Power Company (Different_Generic_Power_Company) was successfully looked up
08/01/2015 01:37:24 AM - Power Company (Different_Generic_Power_Company) was successfully looked up
08/01/2015 01:38:14 AM - Broker (Generic_Broker) was successfully looked up
08/01/2015 01:38:14 AM - Meter (1) was successfully looked up
08/01/2015 01:38:14 AM - Power Company (Different_Generic_Power_Company) was successfully looked up
08/01/2015 01:38:14 AM - Power Company (Generic_Power_Company) was successfully looked up
08/01/2015 01:38:24 AM - Power Company (Generic_Power_Company) was successfully looked up
08/01/2015 01:39:24 AM - Power Company (Generic_Power_Company) was successfully looked up
08/01/2015 01:40:24 AM - Power Company (Generic_Power_Company) was successfully looked up
08/01/2015 01:41:24 AM - Power Company (Generic_Power_Company) was successfully looked up
08/01/2015 01:42:24 AM - Power Company (Generic_Power_Company) was successfully looked up
08/01/2015 01:42:47 AM - Power Company (Generic_Power_Company) was successfully looked up
08/01/2015 01:42:49 AM - Power Company (Generic_Power_Company) was successfully looked up
08/01/2015 01:42:50 AM - Power Company (Generic_Power_Company) was successfully looked up
08/01/2015 01:43:24 AM - Power Company (Generic_Power_Company) was successfully looked up
08/01/2015 01:44:24 AM - Power Company (Generic_Power_Company) was successfully looked up
08/01/2015 01:45:24 AM - Power Company (Generic_Power_Company) was successfully looked up
08/01/2015 01:46:24 AM - Power Company (Generic_Power_Company) was successfully looked up
08/01/2015 01:47:24 AM - Power Company (Generic_Power_Company) was successfully looked up
08/01/2015 01:48:24 AM - Power Company (Generic_Power_Company) was successfully looked up
08/01/2015 01:49:25 AM - Power Company (Generic_Power_Company) was successfully looked up

*Figure 7*