SmartGateway Architecture

The SmartGateway mainly consists of the following components as separate Linux processes:

1. DataManager

2. DeviceManager

3. QueryAgent

4. RuleManager

5. HTTP RESTful Server

The following section delves into the details of each of these components, describing its overall conceptual layout and then moving to their interface APIs.

1. The Data Manager

The Data Manager is responsible for the storage and access to the data that is collected from the various sensors and devices that are deployed in the particular instance of the SmartHome. This component consists of a DevStorage, StorageSlot and the TimeManager. Every Device (or Virtual Device or Data Source) has its associated DevStorage structure and there is a StorageSlot for every command supported by that Device. For example, if we consider the Device to be a Lighting Device, then an instance of it say "lighting1" would have a DevStorage name of the same name and a StorageSlot corresponding to its supported Command "get\_status". Therefore, if we consider a compound Data Source like another SmartHome instance represented as a Device then the SmartHome instance would have a DevStorage corresponding to it and each command (representing its LightingDevice or TemperatureDevice etc) would each have its corresponding StorageSlot.

Each of these StorageSlots represent a particular Storage strategy. Currently, the framework supports two different kinds of storage strategies - In-memory and Cloud storage. The In-memory storage is a simple RingBuffer implementation derived from the popular Boost library called "boost::circular\_buffer". Also, the storage slot maintains a data pertaining to the past 1 hour (3600) seconds. Although this is a configurable C macro, increasing this value is not recommended as the available RAM on the Intel Galileo Development Board is fairly small.

The Cloud Storage capability enables the Data Manager to push incoming streaming data from the Device Manager directly to the designated cloud infrastructure. In this case each storage slot corresponds to a specific "Channel" in Cloud Storage terms. TODO: insert cloud functionalities here.

As soon as new data is received by the DataManager it needs to be timestamped. These timestamps are represented as simple integers and follows the Linux timestamping scheme of "epochs". A separate module known as the TimeManager has been created to maintain a process wide time-stamping counter and avoid the overhead of making a system-call every time we need to use one.

Based on the above stated conceptual layout of the Data Manager, the following APIs have been implemented:

"new" : This command is used to create a new DevStorage compartment within the DataStore. This is invoked as soon as the Device Manager initializes a new device.

"insert": This command is used to insert new streaming data into the DevStorage. This is invoked by the Device Manager when it receives new data from any of its sensors/devices for a particular command. When the data arrives at the DataManager, it is time-stamped and pushed to either the In-memory storage or Cloud-storage as per the configuration. TODO: May also be called by the RuleManager to mark events.

"find": This command is used to query the DevStorage for a particular Device with a particular Command at a particular Timestamp. TODO: mention the different ways to retrieve the data - min, max, avg, median, basically these will also depend on what the cloud storage has to offer.

2. The Device Manager

The Device Manager is responsible for managing access to the actual physical sensors and devices. Every device - physical or virtual - is represented as a separate DBus object within this framework. Therefore, accessing any of the functionality of the these devices means sending across DBus messages. Also, any data generated from these devices are communicated to the other components like the DataManager via DBus messages. A device could be an actual physical device like a temperature sensor or a table lamp or it could be a software component - For example the NotificationAgent that we will see later is actually a represented as a device under the Device Manager or it could another compound device consisting of other smaller devices. For example, we can have another SmartGateway represented as a device under the Device Manager of another SmartGateway thus allowing a hierarchical structure of multiple SmartGateways to allow for a wider scope of application.

DBus requires each DBus object to expose an "Interface Definition" XML file. This requirement very cleanly resonates with the representation of a device. As such, every method defined in the Interface Definition file maps to a specific functionality of the devices. Also, the arguments of these methods correspond to the data required by the devices to implement their functionality in their lower layer userspace driver architecture. To allow for better device discovery and functionality resolution, each DBus object has a service name that corresponds to its Class name with the SmartHome Ontology file. Similarly the Object Path and Interface Path correspond to their respective components in the Ontology. This makes querying for devices very flexible as we can now search for devices via SPARQL or DL-query. This means now we do not need to remember the device instance names to access their functionality. For example, to turn on a light in the living room, one can query the Ontology file based on the functionality of "Light" and location as "LivingRoom". This query if successful would return the exact device instance name that can now be used to access the functionality of turning on or off the lights.

The Device Manager architecture heavily relies on the inheritance and virtual function capabilities provided by C++. On the highest level of abstraction the Device Manager consists of a DeviceAgent that maintains a map of all the devices that have been instantiated and deployed. These instantiation requests arrive from the query agent as the query agent has access to the Ontology file that contains the details of the layout the specific SmartHome.

Each device in the Device Manager is a DeviceBase on the lowest level. The DeviceBase class instantiates 2 pthreads per device - one to read and one to write. There are 2 kinds of reader threads - one reads periodically and one reads only on demand. Most of the common sensors need to be polled periodically to retrieve their measurements, in this case the periodic reader-thread is the most apt. The frequency of polling the actual device is set as configuration during the instantiation phase and is also governed by the Ontology. In fact other configurations are also sent in the instantiation request by the query agent from the Ontology file. This component exposes an API that needs to be overridden by any device driver using this framework. For example, the API contains OnInit(), which must contain the device initialization of its handlers or any other initializations, the Write() to populate the driver buffer with the data that needs to be written and SendToDevice() to handle the actual communication with the device i.e. sending the contents of the write buffer to the device. A similar strategy is used in case of Reading from the device.

The next level of abstraction is the DeviceHandler. This abstraction allows the device driver to communicate using a custom handler. For example, you can have a INT file descriptor for communication over sockets or file io or it could be an OpenCV handle to handle communications with the WebCam or other imaging tools.

Once the low level device handling has been taken care, it needs to establish itself as a part of the whole device network framework. This includes some form of inter-process communication and signalling on the status changes of the devices or any detected errors. The next layer of abstraction achieves this by creating a DBus object that is tied to the lower level device handler for the device. Therefore, any communication with the outside world (i.e the DataManager or the HTTPRest Server or RuleManager) is handled over DBus.

The following steps need to be followed to successfully deploy a new device driver under this driver framework:

1. Create an Interface Definition XML file that describes the different methods and its arguments that the device supports.

2. Use the "dbus-binding-tool" to create the server bindings using the above XML file.

3. Now that we have the server bindings for the device, we need to write code to make these bindings accessible through the DeviceManager Framework. Since this code follows a similar structure as any device, a Code Generator has been written to produce C++ code given the Interface Definition XML file.

4. Once the DBus object has been successfully represented in code, we need to create the C++ files to override the DeviceBase APIs as described in a previous section. This completes the representation of a new device in the framework.

5. Now use the API "DEVICE\_FACTORY\_REGISTER\_DEVICE" exposed by the DeviceAgent to register the newly created device into the framework so that it can be discovered and instantiated. (The instantiation request comes from the query agent using the "new\_device" DBus method call to the DeviceAgent.)

The Notification Agent is a sub-component of the Device Manager that is implemented as a virtual device within the DeviceManager framework.