EECE417 Project Report

Our final implementation of our smart meter remained consistent with the general vision that we outlined in the design doc. The concept of agents acting upon a blackboard remained a strong theme in our architecture. We endeavored to translate every action into something done by an actor. Each actor is an agent with an execute() method that defines its interaction with a shared key-value store called the Blackboard. In our implementation we elaborated on what was vague in our design doc, and also extended our idea of agents to encompass the server as well, leading to interesting possibilities.

Synchronization

One thing we were vague about in the design doc is how synchronization is accomplished in the blackboard data-structure. The underlying implementation is just a java map<String, Object>, which is synchronized for its puts and gets, but the objects in our blackboard are often mutable objects, like `Queue`'s and `ArrayLists`, and we wish to extend synchronization to these.

We accomplish this by maintaining a parallel map of a Mutex like object, called a `BlackboardLease`. A map exists of type `map<String,BlackboardLease>`, and each entry in the Blackboard has a corresponding `BlackboardLease`. A `Blackboard.release()` method was added to allow the release the mutex.

A BlackboardLease differs from a normal mutex in one key way: every call to acquire returns a unique lease password which will be required to later release it. We added this to ensure that other agents were not releasing mutexes that they had not acquired (which would allow them to then acquire them while other agents are possibly mutating the data they protect). Furthermore, this makes sure all releases occur only after acquiring.

The password to release an entry is returned from `Blackboard.Get(String key)` in a tuple of the requested object and the password to later release it with. To release its lease on a mutable object, the agent later `calls Blackboard.Get(String key, String password).`

Fault Tolerance

As stated in the design doc, our agent-blackboard paradigm allows facilities for fault tolerance. The innate independence of the `Agents` allows for their failure to be isolated to the agents that depend upon the results that they produce. Their independence allows them to be run in separate threads (recall that there is a one to many mapping of `AgentThreads` to `Agents`).

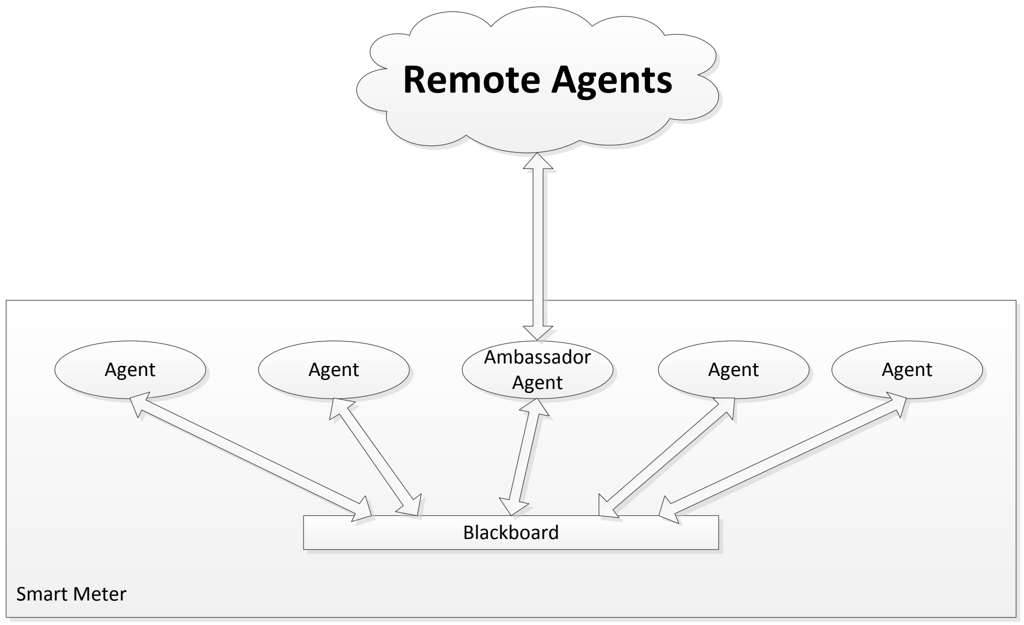
So if one agent hangs, unrelated agents (i.e. agents that do not depend on a result can continue).
In addition to this design feature, we decided to implement a watchdog to deal with hung-up agents. Agents can register on a list held on the blackboard that is checked periodically by a `WatchDog` class (which is a special agent). If an agent does not update a timestamp that it maintains in a blackboard after a certain period of time, the `WatchDog` resets the entire system.

Expansion of the Agent Paradigm

We found it conceptually satisfying, and practical to extend our agent paradigm to the outside world that interacts with our smart meter. The server implements, in our model, a remote agent. It can send GET and SET commands to the smart meter, that act on the blackboard via the `AmbassadorAgent` class. The `AmbassadorAgent` is the proxy for remote agents; it executes the GET's and SET's and returns the results.

In theory it should be possible to implement an arbitrary remote agent that executes via the proxy. Due to limitations on time, the function set implemented for remote agents does not include modifying blackboard entries that are collections (e.g. queues and array lists). This distributed agent scheme gives us complex server-side functionality for free. The server can pull data easily, and arbitrarily.

The below figure summarizes this:



GUI

Our GUI has two components. The first component is a hardware emulator of an 8-segment display that shows time, voltage and current. It has an `Agent`, the `GUIAgent,` which polls the pertinent values in the smart meter, and converts them to the lit up segments on the 8-segment display.

The other component is a server terminal. This displays pushed data sent by the smart meter, and allows for the input of GET pull commands, and SET commands manually which act on the `Blackboard`.

Agent Instantiations

Our design doc did not go into much detail regarding the concrete instantiations of the agent interface that will exist. The follow instances were implements:

- **`AmbassadorAgent`**: Proxy for remote agent (i.e. Server)
- **`ClockAgent`**: Updates the clock entry in the blackboard.
- **`GUIAgent`**: gets necessary values from GUI and displays them on the 8-segment display emulator.
- **`IOAgent`**: handles messages to and from the Server.
- **`PowerAgent`**: calculates mean power and power quality and places on the blackboard.
- **`PowerLossAgent`**: monitors power values in blackboard for power loss to give last breath and first breath alerts to server.
- **`SensorAgent`**: reads a sensor that is passed to it at construction.
- **`TiltSensorAgent`**: monitors the tilt sensor, and sends message to server in case of adulteration.

Missed Requirements

The requirement for encryption, device registration with server, and remote flashing were not implemented. The former can be implemented within our IOAgent, but we felt it would be too much effort for little benefit to the project. The latter two were too difficult to accommodate in our architecture.

Miscellaneous

Various classes were added as utilities to our architecture, and are documented in the javadoc. This includes an enumeration for sensors, GUI classes, and other things.

Challenges

Our agent paradigm is threaded, and highly parallel with dependencies between the various agents. This highly concurrent nature of these interactions lead to many synchronization related debugging issues including deadlocks.

Another challenge was bringing the server into the operational domain of agent-blackboard system. It was difficult to process the asynchronous GET’s and SET’s within our system, and the function set of the remote agent is limited.