VOLTTRON: An Agent Execution Platform for the Electric Power System

Bora Akyol, Jereme Haack, Selim Ciraci, Brandon Carpenter, Maria Vlachopoulou, Cody Tews
Pacific Northwest National Laboratory, Richland, WA, USA 99352
{bora,jereme}@pnnl.gov

ABSTRACT

Volttron is an agent execution platform that is engineered for use in the electric power system. Volttron provides resource guarantees for agents in the platform including memory and processor utilization; authentication and authorization services; directory services for agent and resource location; and agent mobility. Unlike most other agent platforms, Volttron does not depend on a single agent authoring language. Instead, we chose to design and implement Volttron as a platform service and framework that is decoupled from the agent execution environment. A prototype implementation of Volttron has been written in Python (using Python v2.7.2) and we have executed agents written in Python and Java and as shell scripts. The intended use of Volttron is in the power distribution system for managing distributed generation, demand-response, and plug-in electric vehicles.

Categories and Subject Descriptors

I.2.11 [Artificial Intelligence]: Distributed Artificial Intelligence—Multi-agent systems

General Terms

Management, Security

Keywords

Multi-Agent systems, Mobile Agents, Agent Security, Electric Power System, Demand-Response.

1. INTRODUCTION

Volttron is an agent execution platform engineered for use in the electric power system to support mobile and stationary software agents to perform both information sensing and control actions. Devices deployed in the electric power system have to meet very strict requirements for availability, reliability, and security which imposes requirements for agent execution platforms as discussed in greater detail in [3]. To summarize, any agent platform that is used in the electric power system must provide:

Resource guarantees for agents and the underlying platform: Agents in the electric power system may perform both information sensing and control tasks. In order to execute their tasks reliably, the agents require computational resources. If the platform cannot provide guarantees for available processing power, memory, or storage, then the agents may not be able to complete their tasks. On the other hand, if an agent consumes too much memory or processing power, it will adversely impact other agents or even the base platform functions. Therefore, a platform that provides resource guarantees for reliable execution of agents is highly desirable.

Authentication and authorization: To meet cyber security requirements, all agent software that executes in the power system must be validated before execution is allowed. All data carried or produced by agents must be protected against tampering. In addition to these authentication tasks, activities performed by agents must be authorized according to the policies instituted by the electric utilities. For example, a metrology agent may not have a need-to-know to access data in a smart meter related to power quality measurements or network authentication credentials.

Directory services: Directory services are used for access to credentials and for determining nearby resources and devices. Use of directory services allows us to avoid hard-coding information into deployed devices and supports the mobility and assignment of tasks to agents.

An agent mobility service: In order to accomplish their tasks, mobile agents need to be able to move between different devices. We chose to incorporate all aspects such as packaging, transport, determination of destination, and validation of agents into Volttron. This allowed us to keep the agents as simple as possible. Volttron not only provides services for agent mobility but also defines two classes of storage for agents. Mutable luggage is used for agent data collection; immutable luggage is used for configuration such as tasking that the agent needs to carry as it traverses the system. Incorporation of the agent mobility service into our framework also decouples the mobility of agents from the choice of a specific agent execution environment and authoring language. Additionally, an agent's migration path is preserved as part of the immutable luggage.

The research presented in this paper creates a distributed agent-based framework, referred to as *Volttron*, for use in the electric power system. Agents can be used to add "intelligence" to the sensors and controllers used in today's electric power system in an elegant and predictable manner.

Our research is significant because it lays the software platform groundwork for distributed operation and control of the electric power system, especially at the distribution layer where end customers are being served. Unlike other agent platforms, Volttron does not depend on a single agent authoring language. Instead, we designed and implemented Volttron as a platform service and framework that is decoupled from the agent execution environments. A prototype implementation of Volttron has been written in Python (using Python v2.7.2), and we have executed agents written in Python and Java and as shell scripts. The intended use of Volttron is in the distribution system for managing distributed generation, demand-response, and plug-in electric vehicles.

This paper presents the results of an ongoing research project with the goal of introducing our work to a broader community and getting feedback. The rest of this paper is organized into four sections. In Section 2, we describe prior work as it relates to Volttron and present potential use cases and applications to demonstrate the need for our platform. In Section 3, we present the detailed design of our software framework. In Section 4 we focus on agent applications that do or can use Volttron. We present our future work directions in the conclusion.

2. PRIOR WORK

This section describes work related to our Volttron research.

2.1 Platform Software Options

Several technologies initially appeared promising but on further investigation had drawbacks for our needs. One is Squawk VM [18], an open-source platform for wireless sensors. Certified on the Java Micro Edition Information Module, it provides developers with a standardized Java language. A major advantage of Squawk is that it can run directly on hardware without an operating system. In Squawk, the entire state of an application (treated as an isolate) is stored as java objects that can be serialized to disk/stream. However, Squawk had stability issues and limitations in development environment and capabilities.

Another initially promising technology is JADE (Java Agent Development Framework). JADE is a software framework fully implemented in Java [4]. It provides a set of Java classes that allow a developer to build a FIPA-compliant multi-agent system quite easily. However, resource management, an important requirement for our platform, is not strongly supported. We also questioned how well supported JADE currently is because activity on the JADE website and mailing list seems to have dropped off.

In general, we were unable to find the combination of security, scalability, and resource management we needed in other agent frameworks. This led us to write specifications for our own platform to handle these issues and provide a language-agnostic environment to run existing agent frameworks on top of ours.

2.2 Agent-Based Solutions

Software agents provide a powerful method of addressing the scalability and resilience issues inherent in the power grid as discussed in [8] and [15]. Several existing projects use this paradigm for various use cases in the power grid, including [16], which uses a community of different agent types to diagnose power system faults.

Many of these projects detail an approach for using agents, but their demonstration is limited to simulation or proof-of-concept implementations that would not operate well in the field. Our research addresses these problems by providing a platform that handles the security and resource constraints and allows other researchers to focus on the operation of their agents. Therefore, any previous work done in the agents field can be accommodated by our platform. We illustrate this in Section 4 with examples of previous agent systems.

AgentScape [20] was in development at the same time as this project and provides many interesting parallels with our research. However, it does not discuss resource management, which is important in the power system environment. We will be investigating AgentScape further.

3. VOLTTRON PLATFORM

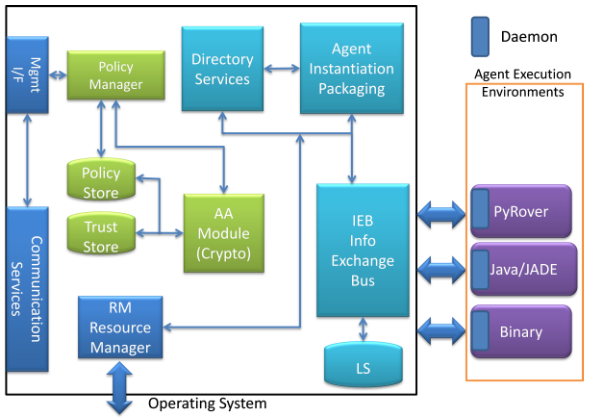

Figure 3-1: Volttron Platform Components

3.1 Architecture and Design

This section discusses the architecture and design details of the Volttron agent execution platform. The platform building blocks are shown in Figure 3-1. The agent execution platform (AEP) exists between the operating system and the agent execution environments. As shown in the figure, Volttron supports multiple agent execution environments (AEEs) such as Java, Python, and platform-specific binary objects. The Volttron platform consists of communications services (CS); resource manager (RM); authentication and authorization (AA); directory services (DS); agent instantiation and packaging (AIP); and information exchange bus (IEB) modules. The AA module includes a policy and trust store as well as an optional policy manager function. The IEB module includes a local store (LS) to provide non-volatile storage for agents. To summarize, AIP is responsible for packaging, instantiation, and coordination of agents' movement. The AA module provides validation of agent payloads, authenticates peer platforms, and handles public and private credentials. The DS module provides name, resource, and public credential to location and network identity mappings. RM is the gatekeeper for the platform. It decides if the platform has enough resources left to accept the execution of an agent. RM also manages access controls for AEE "containers." Finally, RM monitors use of resources and either warns or terminates misbehaving agents. The CN module is responsible for reliable and secure transfer of packaged agents and peer-to-peer communication between Volttron platforms.

Volttron is currently implemented in Python v2.7 and leverages many existing Python modules developed by the open source community. In the remainder of this section, we discuss Volttron module details focusing on functionality and interfaces.

3.1.1 Agent Instantiation and Packaging

The AIP module controls the workflow for sending and receiving Agent Transport Payloads (ATP) including their creation, extraction, and interpretation. ATP contents are illustrated in Figure 3-2 and include the agent code, the agent execution contract, immutable, and mutable luggage (read-only and writable files). In Section 3.1.2, we describe the Scope of Influence (SOI) concept and also discuss the signatures shown in the figure. AIP extracts the contents of an ATP and places the contents of the luggages in the virtual file store in a location where the agent can access them.

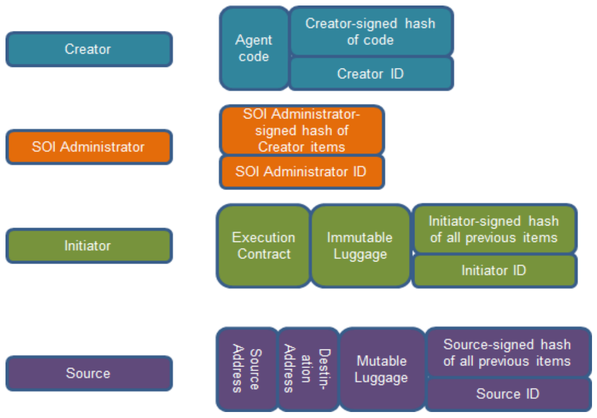

Figure 3-2: Agent Transport Payload

AIP also controls and implements the main workflow of agents entering and leaving the platform. An agent payload enters the platform via the communication module and is sent to the AIP module. The AIP module then performs security and resource checks. The authorization and authentication information in the payload is passed to the AA module (as detailed in 3.1.2). If verified by the AA, the resource requirements of the agent, detailed in the Execution Contract, are then passed to the Resource Manager (section 3.1.3). If sufficient resources are available, then the Resource Monitor returns a process ID for the agent to track resource usage. If either of these checks fails, then the agent is rejected and an error notification is returned to the sending platform. If these checks pass, then the AIP extracts the agent code, immutable luggage, and mutable luggage. The immutable luggage contains read-only information, such as configuration files, provided by the Initiator. The mutable luggage is a dynamic payload that the agent brings with it to each host, which helps the agent maintain state (these are the equivalent of the Agent Containers in AgentScape [20]). After extraction, the AIP notifies the AEEManager, which will then start the agent in the appropriate Agent Execution Environment with the Resource-Manager-supplied process ID.

After an agent finishes its task on a platform, it uses the Directory Service to find an appropriate platform to move to based on an itinerary or by searching for certain properties. Upon finding a target, it requests to move via the IEB which provides a mechanism for agents written in any language to communicate with the rest of the platform. When the AIP receives an agent movement request, it re-packages the agent by using the same agent code and immutable luggage contents it received but adding the updated mutable luggage. AIP then uses the CS Module to send this payload to the target platform.

3.1.2 Authentication and Authorization

When considering the security properties of confidentiality, integrity, and availability, electric power system utilities place the highest priority on availability and the lowest on confidentiality. The security property of integrity (source integrity/identity and data integrity) contributes to reliable operation by minimizing the risk of system compromise and by allowing detection of system compromise that does occur. The AA module directly addresses integrity while providing the infrastructure for confidentiality. For the remainder of this paper, we assume that devices using Volttron have sufficient computing resources to perform cryptographic authentication, authorization and trust functions, including asymmetric cryptography used for identification and integrity. Note that even low-cost micro-controllers can now be obtained with hardware cryptography support; therefore, this is not a burdensome requirement.

3.1.2.1 Usage Scenario

In Section 3.1.1, we discussed how the AA module is used as part of the workflow of moving agents across the electric power system. The primary function of the AA module is providing cryptographic integrity and authentication services to other modules in the Volttron platform. The AA module performs this task by using public key cryptography (specifically X509v3 certificates [6]); but instead of a strictly hierarchical system, it allows for a flatter trust model, which will be described in the next section. For most commonly seen operations, the public certificates are retrieved via the directory services module, which also alleviates the need for an explicit revocation model that uses a certificate revocation list [9]. Our approach is similar conceptually to online certificate status protocol with certificate stapling [7]. It is important to note that Volttron provides integrity services for agent code; agent tasking and configuration; and data carried by the agents.

3.1.2.2 Scope of Influence

The electric power system follows a fairly hierarchical organizational structure. A country is divided into regions. In each region (*simplistically*), there are balancing authorities, regional transmission operators, power generation suppliers, and distribution utilities. Within a distribution utility, power flows through distribution lines to distribution substations and then to feeder lines that deliver power to customer premises. Hence, the intelligence and distributed computing capabilities used in the electric power system must follow an organizational structure close to the way power flows through the system. With this in mind, we have defined ***SOI*** as an organizational boundary within which a set of software agents can communicate, cooperate, and exchange information. Each SOI has a unique identifier.

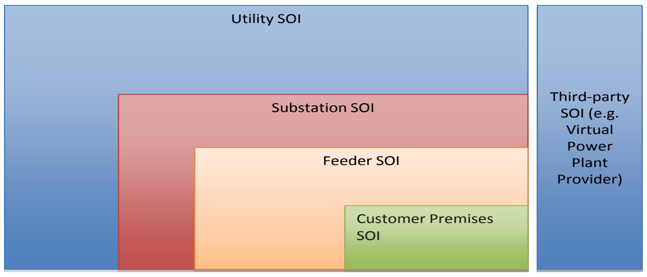

Figure 3-3: Scopes of Influence

An SOI has one or more non-human ***initiators*** (i.e., computing entities) that are responsible for assigning tasks and dispatching agents. For this project we have defined a hierarchical structure composed of five SOIs as shown in Figure 3-3. Within a utility SOI, there are multiple substation SOIs. Within a substation SOI, there are multiple feeder SOIs. Within a feeder SOI, there are multiple customer premises SOIs. Note that while the utility, substation, and feeder SOIs may be under the same administrative authority, the customer premises SOI is more than likely under a different administrative authority; i.e., the customer. The third-party SOI represents entities that are not utilities and not owned by a utility that perform functions within the power system. An example of a third-party SOI is a virtual power plant provider. Note that a customer may choose to trust a third-party SOI, even if the utility serving the customer does not. The basic properties of the SOI are summarized below:

1. An SOI consists of electric power system devices (referred to as “devices” in the remainder of the text) that are capable of hosting software agents.
2. Each SOI has at least one ***initiator.*** The SOI initiator serves as the trust root for the SOI. If the initiator becomes unreachable, then a new initiator can be selected and all devices shall continue to operate with credentials cached in their local trust store. Based on an SOI’s trust policy, an initiator can extend trust to an external entity by signing its credentials to form a trust chain very similar to PGP [1].
3. The human administrators of an SOI can upload/enable agents to be run within an SOI. Each SOI has a policy for determining whether an agent is authorized to run within an SOI. This policy is distributed to all devices within an SOI. An SOI may ban all external agents or it may choose to authorize agents received for execution from other SOIs through policy-based trust negotiation.
4. Initiators are responsible for assigning tasks and dispatching agents. Each SOI has at least one ***initiator.*** If there is more than one initiator in an SOI, it is assumed that the initiators collaborate to choose a master initiator with all other initiators in the SOI serving the master initiator. The initiators are provisioned by the administrator of the SOI.
5. Initiators are also responsible for providing ancillary services to the agents within an SOI. The initiator’s ancillary services may include (but are not limited to) agent discovery and directory services, trust services (e.g., reputation management and trust negotiation), and communication between this SOI and its parent SOI. An initiator may also participate in network services such as routing and name resolution.
6. The task assignment is performed based on the agent capabilities. An initiator has a ***task list*** (signed by an SOI Administrator) that contains a set of tasks that need to be performed. For example, a task at a feeder SOI may be to check power usage to determine whether there is theft of power on that feeder. The task list at the Initiator may include tasks that need to be performed periodically. A task may be composed of other tasks. The ***agent library*** includes a catalog of agent metadata that are organized by task in order to choose an appropriate set of agents to perform each task. An agent may be mobile or stationary. Section 3.1.1 describes the process by which the Initiator prepares an agent for deployment.

Identification Credentials:

Volttron supports identification of core entities in the system: SOI organizations, SOI administrators, devices, creators, and initiators.

- Each SOI Organization will possess an X.509 key pair (private key and public certificate) intended for identification and signature use and signed by a recognized certificate authority. A utility may choose to be its own trust root and operate a CA.
- Each SOI Administrator will possess an X.509 key pair intended for identification and signature use and signed by the SOI-owning organization. The certificate will identify the owning organization.
- The Initiator will possess an X.509 key pair signed by the SOI Administrator.
- Each device on which an Agent Platform is installed will possess a network address and an X.509 key pair. Only one Agent Platform will be installed per device and they will share the same identification. The IP address and the X.509 certificate and public key will be available to other entities via Directory Service lookup.
- Each Agent Creator will possess an X.509 key pair intended for identification and signature use and signed by a certificate authority recognized by the SOI Administrator.

Each deployed agent will possess an identifier composed of (or mapped to) the company name, software name, software version, and an instance ID.

3.1.2.3 Authentication and Authorization Module Functions

The AA module provides integrity services to the framework including identification services, source integrity, and agent transport payload integrity. As described in the AIP workflow in Section 3.1.1, the AA module is mainly called by AIP and DS modules. To perform its functions, the AA module communicates with peer AA modules on other devices that are part of the SOI. The AA module also uses directory services to retrieve credentials of devices within an SOI. The AA module provides programming interfaces to compute a cryptographically signed hash; to verify a signed hash received from a peer; to validate requested privileges against configured platform policies; and to perform credential look-ups using directory services.

3.1.3 Resource Manager

The RM module is responsible for controlling resources assigned to an agent process and limiting the use of those resources based on the contract presented during agent instantiation. Resources that may be controlled by the RM include, but are not limited to, CPU, memory, and I/O devices. An agent that is detected consuming resources above the contracted amount will be subject to termination if it fails to correct the action upon notification from the RM. The RM may use kernel-level, operating-system-dependent methods to implement the required functions.

3.1.3.1 Usage Scenario

An agent arrives on the platform requesting execution. It passes validation checks and is unpacked and prepared for execution. Before reserving the required resources, a check is made against static capabilities and resources to ensure the platform is capable of supporting the agent. If the platform can support the agent, then dynamic capabilities are checked and an attempt is made to reserve the required resources. If the resources are available and successfully reserved, the agent is free to execute in the reserved environment. If the RM detects that the agent is operating outside the bounds of the execution contract, then the RM attempts to notify the agent of the contract breach and give it sufficient time to correct the situation. If the agent fails to correct the breach of contract in a timely manner, the RM terminates the agent. In our prototype implementation, RM manages the following resources: CPU (CPU sets, affinity, maximum utilization), memory (maximum utilization), I/O devices (access [read/write], maximum utilization).

3.1.3.2 Agent Execution Contract Definition and Negotiation

The agent execution contract defines the execution agreement between the agent and the platform. The agent agrees to provide some service while executing within the bounds it established with the platform and the platform agrees to provide the agent resources as long as it does not abuse or misuse the platform or the leased resources. Abuse and misuse can be difficult to detect, but the platform will make its best effort and reserves the right to terminate misbehaving agents.

Like most contracts, some things are negotiable and some are not. Non-negotiable items may include the hardware configuration and other properties that may prevent the agent from executing its negotiating code. Negotiable items are those things that may change dynamically or that may not prevent an agent from executing. Non-negotiable items should be defined statically and in a common format that can be read, checked, and given a simple response. Once it is established that an agent has the ability to execute in a restrictive environment, a negotiation phase is entered where the agent and the platform are allowed to banter back and forth to form an agreement. Either party may terminate negotiations at any time.

Contract terms may be set for three classes of items: authorization, capabilities, and resources. Authorization may be thought of as a specialized capability. An example of authorization is an agent requesting to run with elevated privileges or to open a privileged port. Capabilities cover the set of features or services the platform, operating system, or other software may provide. Resources are physical or virtual devices on the platform. Capabilities can be broken down further into two types: hard and soft. A hard capability is one that is unlikely to change without a modification to the system that would require the platform to stop hosting agents (e.g., a change requiring a reboot). Soft capabilities are those that are negotiable or that may change without stopping the platform service. Likewise, resources can be static or dynamic. As the platform evolves, there will be different versions of the platform software as well as the agent execution environments. In a network, multiple versions of the platform framework will exist. Therefore, when an agent moves to a new platform, part of the validation is to ensure that the versions of the AEE and the platform are compatible with this agent.

The contract must be in a format that is expressive enough to define the requirements of the agent and the platform while flexible enough to allow for new features and for skipping items the agent or platform either does not understand or does not support. A text format is best suited for this as binary formats are fragile in the face of change. Therefore, the selected format of the contract is an RFC 2822 message [11] supporting MIME multi-part message bodies (RFC 2045 [12]) and non-ASCII header extensions (RFC 2047 [17]). Requirements are set in the message headers and may point to sections in the body for additional flexibility.

3.1.3.3 Implementation on Linux 2.6.x/3.x

The interface above may be implemented on a Linux system using a combination of cgroups, Linux Containers, Linux capabilities, procfs, sysfs, fork, and exec. Data needed for *get_static_resources()* and *check_resources()* could be gathered from procfs, sysfs, and system calls. Agent environments would be reserved by calling *reserve_resources()*, which would likely fork, closing all files; attempt to create a Linux container, which uses cgroups; set the appropriate capabilities; drop privileges; and return the child process ID as the *reservation_id*. *exec_resource()* would cause the process to exec the given executable with the given arguments and environment, which could be passed to the process using a UNIX domain socket or set in the virtual file system. cancel_reservation() would notify the process to continue without giving it an executable, causing it to terminate without performing further actions.

3.1.4 Communication Services

The CS module provides a remote procedure call (RPC)-based communication channel to other devices in an SOI as well as to the initiators of other SOIs. We rely on using the TCP/IP stack built into our operating system choice (Linux). All platform modules communicate with their peers using the RPC mechanism provided by the CS module. The communication integrity and confidentiality are provided by SSH using public keys. Just like the X509 certificates used by the AA module, the SSH public keys are retrieved by querying the directory services module. To avoid a circular dependency, a small subset of security credentials are pre-configured for each device as part of its provisioning and enrollment. In our Python implementation, we use paramiko [5] for SSH and bjsonrpc for RPC [13].

3.1.5 Information Exchange Bus

The IEB provides a method for agent-to-agent and platform-to-agent communications. It also provides for local storage and retrieval of persistent and temporary data. Agents can communicate with other agents on the same platform or on remote platforms using a topical publish/subscribe pattern for the communications. Each agent is automatically subscribed to a platform-to-agent topic that allows the resource monitor to send notifications. Since file operations are provided by most programming languages, the IEB implements a publish/subscribe system using a virtual file system to ensure maximum flexibility for interacting with the agents.

Agents do not communicate directly with the IEB Topic Manager. AEEs have an API for the agents to call into to subscribe. AEEs go through the AEEManager to talk to the IEB, which returns a topicID/Filename to the agent. The agent then works with the file directly or through an AEE-specific API.

3.1.6 Agent Execution Environments

AEEs are where the agents are actually run and can be implemented in a variety of languages and frameworks. AEEs are specific to the environment agents require. Our initial implementation provides an AEE for Java, Python, and executable as examples for building additional environments.

Each AEE needs to communicate with the AEEManager in order to enable receiving/sending an agent. They also route agent requests to the AEEManager via the IEB so that modules in the platform have a single point of contact with the agents.

When an agent is created, it can be given a standard topic that is monitored by the AEE. The agent makes movement requests, gets notified of state changes, etc. via the topic.

3.1.7 *Directory Services*

The DS module allows an agent to dynamically discover capabilities (e.g., available software libraries and hardware sensors) of the devices in the electric power system. Typically, an agent starts this discovery scenario by sending the list of required capabilities to the DS. The DS, in turn, executes a query and returns the list of potential nodes that can host the agent. Using this list, an agent decides on a node and issues a transfer request.

To realize the scenario described above, the DS module should support the following functional and non-functional (FR/NR) requirements (*FR* stands for f*unctional requirement*, and *NR* stands for *non-functional requirement*):

- FR 1: A publishing mechanism that allows the platform modules to announce/denounce capabilities and their network addresses.
- FR 2: A name resolution system for translating the names of sensor nodes to up-to-date network addresses.
- FR 3: A mechanism for querying capabilities of a specific node.
- NR 1: Ability to provide discovery regardless of isolations: due to disconnections, some sensor nodes can become isolated. When this happens, the agents should still be able to discover the capabilities of these nodes using the DS module.
- NR 2: Ability to work independent of underlying networking hardware.
- NR 3: Provide scalable publishing/querying.
- NR 4: Support secure communications.

There are many different technologies that can address these requirements. We chose to focus on Lightweight Directory Access Protocol (LDAP), Simple Service Discovery Protocol (SSDP), and Distributed Hash Table (DHT) because they are publically available and are used successfully in software systems with similar requirements. To help determine which technology to use, we conducted a trade-off analysis among these three technologies. In this section we first discuss the trade-off analysis and how we decided on the technology. We then discuss the directory service implementation.

3.1.7.1 *Trade-off analysis*

For the trade-off analysis, we have conducted a survey on three technologies and categorized their support for the requirements into the following: i) *supported*, the requirement is supported without modifications to the existing implementations/protocols; ii) *Supported with minor modifications*, minor extensions are required to the implementation/protocol for realizing the requirement; and iii) *Not Supported*, the technology is either not designed for the requirement or major modifications are required to the implementation/protocols. Table 1 summarizes the categorization of different technologies.

Table 1 Comparison of directory service technologies

Technology	FR1	FR2	FR3	NR1	NR2	NR3	NR4
LDAP	+	+	+	-	+	-	+
SSDP	+	*	*	+	-	*	*
DHT	+	+	*	*	+	+	+

Legend: + supported, *supported with minor modifications, - not supported

LDAP is a protocol for accessing directory services. The LDAP specification defines an entry of a directory as a set of attributes with a unique *distinguished name*. The entries are stored in a tree-like structure; users can specify what information is required in an entry through schema definitions. LDAP is tuned for client-server architectures, although it is possible to distribute the tree structure among different servers or have multiple servers that synchronize periodically. Due to the client-server nature of LDAP, it does not easily meet the isolation and scalability requirements of Volttron. To address the isolation requirement, we need to execute more than one LDAP server in an SOI and have them synchronize; this in turn requires a lot of synchronization messages and reduces the scalability.

SSDP allows nodes in a local area network (LAN) to advertise presence and network service information. SSDP does not define how a node stores (for querying later) these advertisements. Instead, the protocol focuses on advertisement exchange without server-based configuration: SSDP advertisements can be exchanged without configurations. SSDP achieves this by using multicast in LANs. SSDP does not have the ability to operate in non-LAN environments.

DHTs [2] are decentralized overlays that provide store, delete, and lookup operations similar to hash tables. Typically, the DHT store/lookup operations have a routing depth of *O(logn)* increasing the scalability in terms of overlay size. As DHTs form their own overlays, they can operate with different link-layer protocols. Most DHTs also provide mechanisms to compensate for isolations/connection interrupts. DHT implementations are mainly based on remote procedure calls, making it easy to implement extensions and provide secure communications.

Due to their scalability benefits and flexibility in accommodating our isolation requirement, we decided to use a DHT-based technology. Specifically, we have chosen Kademlia DHT [14] as it already supports data replication to compensate for disconnected (isolated) nodes and the XOR-based distance calculation in Kademlia is suitable for embedded systems.

3.1.7.2 *Directory Services Implementation*

Our implementation is based on the Entangled library, an implementation of Kademlia in Python. Each capability maps to *l* keys from the range of keys associated for capabilities. The capability mapping is pre-defined and stored in each sensor node (i.e., the same mapping is used by each node). To publish a capability, the sensor node first retrieves the *l* keys corresponding to the capability from the map. Then, it issues a store request to the Kademlia network for each of these keys. Here, each store request contains the key and network address of the sensor node. We associate more than one key per capability for replication; in this way, more sensor nodes will store the same value, thus increasing its chances to be found in isolation conditions (e.g., when a set of nodes becomes isolated from the Kademlia DHT).

The store operation in Kademlia locates *k* nodes that are closest to a given key l_1. In our system, this might cause a publish request for a capability c_1 originating from an SOI s_1 to be stored at another SOI. We limit the publication to an SOI by dividing the key space as shown in Figure 4. The first 40-bits are assigned to the unique identifiers of the SOI's. Each sensor node contains a mapping from SOI name to 40-bit SOI key in their local stores.

Before joining the DHT, the DS is given the name of its SOI and the network address of another node in this SOI. With this information, the service first forms its unique identifier key l_n, where the first 40-bits are the SOI key and the remaining 120 bits are randomly assigned. Then, the node issues a publish request for l_n , *node-nam,* allowing the node to join to an SOI.

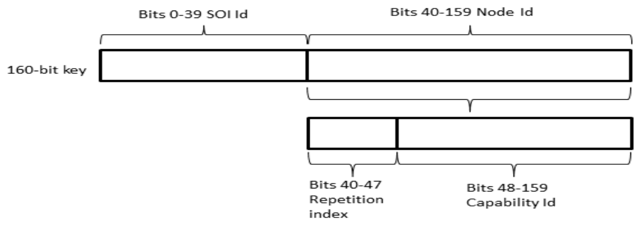

Figure 4. The distribution of the DS key space

Similar to SOI identifiers, each capability is assigned a pre-defined 112-bit key. These keys are stored at the local stores of the sensor nodes. Before a publish capability operation, the service first retrieves the keys associated with the SOI and the capability from the map. Then it forms the key for the capability where bits 0-39 are the SOI key, bits 40-47 are the *repetition index* of the capability, and bits 48-159 are the key of the capability. We programmed the directory service such that it publishes a sensor node's capability more than once in order to provide better tolerance against isolation. The *repetition index* is a counter that is used for both generating a unique key for each repetition and identifying which repetition for a capability is accessed.

For a lookup operation, the agent supplies the directory service name of the SOI and the capability to search. The directory service, in turn, forms the key for the first repetition of the capability (*repetition index=0*). The service then initiates the lookup operation. If the lookup operation does not yield any results, the service increments the repetition index, forms the new key with this index, and restarts the lookup operation. By default, only three repetitions for a capability can be published; hence, the search terminates after three look-ups. The value of the repetition index is configurable.

4. APPLICATIONS USING VOLTTRON

Volttron is a platform that is useful in many contexts within the electric power system. Agents deployed on Volttron can be used to manage resources in the distribution system, increase situational awareness in the transmission system, and diagnose faults in the bulk generation system. A series of agent applications were discussed in previous ATES workshops (http://users.ecs.soton.ac.uk/acr/ates2011/) and certainly will be discussed in ATES2012. We will therefore pick two applications and discuss how they can be implemented using Volttron.

4.1 Plug-in Electric Vehicle Charging

Our first example application was published in ATES2010 and is concerned with the management of charging of plug-in electric vehicles (PEVs) [19]. In this application, a multi-agent system (MAS) is compared to centralized optimal scheduling for managing charging of PEVs. To summarize, the MAS depicted in [19] relies on two types of agents: transformer agents that want to smooth out the load in the feeder and prevent overloading and PEV agents that control the charging of the vehicles. The PEV agents signal their intentions to the transformer agent. The transformer agents then determine the maximum allowed power consumption for each transformer by using peer-to-peer negotiation. Once the maximum allowed power consumption for each transformer is determined, the transformer agents publish the agreed charging power and schedule information to the PEV agents. The MAS depicted in [19] also allows for charge energy reservation requests from PEVs and requires periodic refresh of PEV agent reservations.

We now map the functionality required by this MAS scenario to Volttron capabilities. Volttron provides directory services so that the transformer and PEV agents can discover each other dynamically by searching for each other. Once the discovery process is completed, secure communication channels are established by the provided communication services module. If data are collected via a PEV data collection agent, then the collected data can be packaged with the data collection agent and transported securely using the mutable luggage services provided by Volttron. Additionally, on the same transformers, we may have other agents that are managing the demand patterns of other high power-draw appliances. Resource management capabilities built into Volttron ensure that all agents have appropriate resources to execute their tasks. If a programming error or malicious intent causes an agent to consume an unreasonable amount of resources, then the Volttron resource manager will detect and take appropriate actions to restore fairness. If we were to extend the concept proposed in [19] to allow for communication between the PEV agents directly collaborating to accomplish their goals, then we can use a mobile agent that transfers information between the different PEVs. This mobile agent will take advantage of both immutable and mutable luggage functionality as well as the integrity services provided by our platform. Finally, Volttron supports multiple agent execution environments that allow MAS developers to not be constrained to a single authoring language.

4.2 Fault Location

Our previous example used mainly stationary agents to control the charging of PEVs. In this example, we discuss an application that solves an important problem in electric power system operations: The location of distribution-level fault by leveraging agents embedded within power system devices. The use of MAS in the protection of a shipboard power system was demonstrated in [10] as a distributed network of handheld computers connected to an emulated power system. The authors discuss a method by which MAS can detect and diagnose faults within the shipboard power system and provide an example of a MAS-based system reconfiguration. Extending MAS fault detection and diagnosis to a terrestrial power system introduces the additional constraints of geographically disperse resources, varying communication technologies, and multiple state holders which Volttron can accommodate.

Assuming that the devices (hosting the agents) maintain low data rate communications over the same wires that are used for transporting electricity, we can envision a scenario where a utility would like to know the location of a fault condition on the distribution network. To accomplish this goal, a utility operations center (UOC) will task agents to travel between intelligent electric devices and collect both the path and the device information and bring it back. The collected information can then be imported into UOC systems to find devices that no longer have a communication path. Characterizing the devices without communication can provide an automated method of zeroing in on the common effected area, and the fault location. In this example, all capabilities of Volttron are used. As they jump between devices, agents make use of directory services to decide where they want to go next. AIP module is used to transport agents between devices. Agent code integrity is provided. Agent configuration and tasking orders are stored inside the immutable luggage, and collected agent data are stored inside the mutable luggage. Resource management is used to ensure that agents have adequate resources. Finally, secure communication channels are provided by the CS module.

5. CONCLUSION AND FUTURE WORK

Volttron is an agent execution platform that is engineered for use in the electric power system. Volttron provides resource guarantees for agents and the platform including memory and processor utilization; authentication and authorization services; directory services for agent and resource location; and agent mobility. Unlike other agent platforms, Volttron does not depend on a single agent authoring language. Instead, we chose to design and implement Volttron as a platform service and framework that is decoupled from the agent execution environment. A prototype of Volttron has been written in Python v2.7.2, and we have executed agents written in Python and Java and as shell scripts. Some of our design choices were independently confirmed by another project that was developed within the same timeline as our project [20]. As part of Volttron development, we have also created a trust model that is suitable for use in the electric power system. In the near future, we will be publishing Volttron source code as opensource and looking for opportunities to work with other developers in the MAS field to port their applications to run on Volttron. We expect that capabilities similar to Volttron will be integrated into devices that are either part of or interact with the electric power system.

6. REFERENCES

[1] A. Abdul-Rahman. The pgp trust model. *EDI-Forum: The Journal of Electronic Commerce*. 10(3): 27-31, 1997.

[2] S. Androutsellis-Theotokis and D. Spinellis. A survey of peer-to-peer content distribution technologies. ACM Computing Surveys, 36(4): 335-371, 2004.

[3] Anonymous, 2011.

[4] F. Bellifemine, A. Poggi, and G. Rimassa. JADE: A FIPA2000 compliant agent development environment. Proceedings of the Fifth International Conference on Autonomous Agents. ACM, New York, 2001.

[5] R. Pointer. The paramiko project website. Accessed February 2012 at http://www.lag.net/paramiko/

[6] W. Burr, N. Nazario, and W. Polk. A proposed federal PKI using x.509 v3 certificates. National Institute of Standards and Technology. 1996. Accessed February 2012 at http://csrc.nist.gov/nissc/1996/papers/NISSC96/paper042/pkipap1.pdf

[7] R. Deacon and R. Hurst. Internet Engineering Task Force. RFC 5019, The lightweight online certificate status protocol (OCSP) profile for high-volume environments. 2007. Accessed February 2012 at http://www.rfc-editor.org/rfc/rfc5019.txt

[8] G. Heydt, C. Liu, A. Phadke, and V. Vittal. Solution for the crisis in electric power supply. IEEE Computer Applications in Power. 14(3), 22-30, 2001.

[9] R. Housley, W. Polk, W. Ford, and D. Solo. Internet Engineering Task Force. RFC 3280, Internet x.509 public key infrastructure certificate and certificate revocation list (CRL) profile. 2002. Accessed February 2012 at http://www.rfc-editor.org/rfc/rfc3280.txt.

[10] K. Huang, S. Srivastava, D. Cartes, and L. Li. Agent solutions for navy shipboard power systems. Proceedings of IEEE International Conference on System of Systems Engineering. San Antonio, TX, pp. 1-6, 2007.

[11] Internet Engineering Task Force. P Resnick, ed. RFC 2822, Internet message format. 2001. Accessed February 2012 at http://www.rfc-editor.org/rfc/rfc2822.txt

[12] Internet Engineering Task Force. N Freed and N Borenstein, eds. RFC 2045 Multipurpose Internet Mail Extensions (MIME) Part One: Format of internet message bodies. 1996. Accessed February 2012 at http://www.rfc-editor.org/rfc/rfc2045.txt

[13] D. Martinez. The bjson project website. Accessed February 2012 at http://deavid.github.com/bjsonrpc/

[14] P. Maymounkov and D. Mazieres. A peer-to-peer information system based on the xor metric. Peer-to-Peer Systems: Lecture Notes in Computer Science. 2429: 53-65, 2002.

[15] S. McArthur, E. Davidson, V. Catterson, A. Dimeas, N. Hatziargyriou, F. Ponci, and T. Funabashi. Multi-agent systems for power engineering applications—Part I: Concepts, approaches, and technical challenges. IEEE Transactions on Power Systems. 22(4), 1743-1752, 2007.

[16] S. McArthur, E. Davidson, J. Hossack, and J. McDonald. Automating power system fault diagnosis through multi-agent system technology. Proc. of the 37th Annual Hawaii International Conference on System Sciences, Big Island, HI, 2004.

[17] K. Moore. Internet Engineering Task Force. MIME (multipurpose internet mail extensions) part three: Message header extensions for non-ASCII text. 1996. Accessed February 2012 at http://www.rfc-editor.org/rfc/rfc2047.txt

[18] D. Simon and C. Cifuentes. The squawk virtual machine: JAVA™ on the bare metal. Proceedings of 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications. ACM, New York, 2005.

[19] S. Vandael, N. Boucke, and T. Holvoet. Decentralized demand side management of plug-in hybrid vehicles in a smart grid. Proc. of the First International Workshop on Agent Technologies for Energy Systems, Toronto, pp. 67-75, 2010.

[20] M. Warnier, M. Oey, R. Timmer, and F. Brazier. Enforcing integrity of agent migration paths by distribution of trust. International Journal of Intelligent Information and Database Systems. 3(4): 382-396, 2009.